# 1 Introduction

But :

— permet d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet n'est donc pas connue par l'appelant.

Cas d'utilisation : utilisez le *design pattern* « Fabrique » :

— Les fabriques sont utilisées dans les toolkits ou les frameworks, car leurs classes sont souvent dérivées par les applications qui les utilisent.

— Des hiérarchies de classes parallèles peuvent avoir besoin d'instancier des classes de l'autre.

Diagramme de classes : voir Fig.1

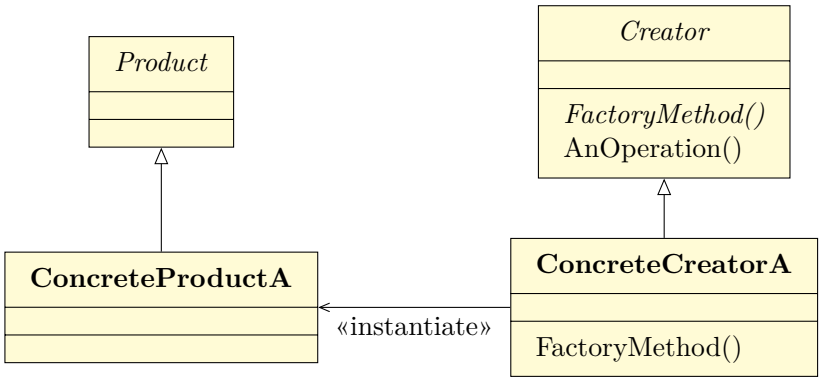


Figure 1 – Fabrique (*Factory Method*) : diagramme de classes générique

Explications de la figure 2 :

— `Creator` :
  — classe abstraite [1] (italique en UML) qui servira d'intermédiaire
  — déclare la méthode fabrique, qui renvoie un objet de type Product. Le créateur peut également définir une implémentation par défaut de la méthode usine qui renvoie un objet ConcreteProduct par défaut
  — peut appeler la méthode `FactoryMethod()` pour créer un objet Product

— `ConcreteCreator` :
  — classe concrète à laquelle on devra *override* la méthode de *FactoryMethod()*, et ajouter dynamiquement des responsabilités (champs, méthodes) supplémentaires.

— `Product` :
  — classe abstraite [2] qui servira d'intermédiaire

— `ConcretProduct` :

1. classe abstraite : classe ne pouvant pas être instanciée.
2. classe abstraite : classe ne pouvant pas être instanciée.

— classe concrète à laquelle on désire ajouter dynamiquement des responsabilités (champs, méthodes) supplémentaires.

— utilisation :

```
/// <summary>
/// The 'Creator' abstract class
/// </summary>
abstract class Creator
{
    public abstract Product FactoryMethod();
}
```

```
/// <summary>
/// A 'ConcreteCreator' class
/// </summary>
class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod()
    {
        return new ConcreteProductA();
    }
}
```

```
/// <summary>
/// The 'Product' abstract class
/// </summary>
abstract class Product { }
```

```
/// <summary>
/// A 'ConcreteProduct' class
/// </summary>
class ConcreteProductA : Product { }
```

— L'objet `Creator` défini la méthode fabrique qui va être utilisé par les interfaces et retournera le `Product`.

— L'objet `ConcreteCreator` *override* la méthode et instentie le `ConcreteProductA`.

— L'objet `Product` qui est une classe abstraite [3].

— L'objet `ConcreteProductA` qui va implémenter l'interface de `Product`

# 2 Exemple

## 2.1 Cahier des charges

Création de différent documents (*Document*) :

— CV :
  — CompétencePage
  — EducationPage
  — ExpriencePage

— Rapport :
  — IntroductionPage
  — ResultatsPage
  — ConclusionPage
  — SommairePage

3. classe abstraite : classe ne pouvant pas être instanciée.

— BibliographiePage

Application (console) permettant :

— de créer un document ;
— d'afficher le nom du document et les noms des pages ;

## 2.2 Diagramme de classes

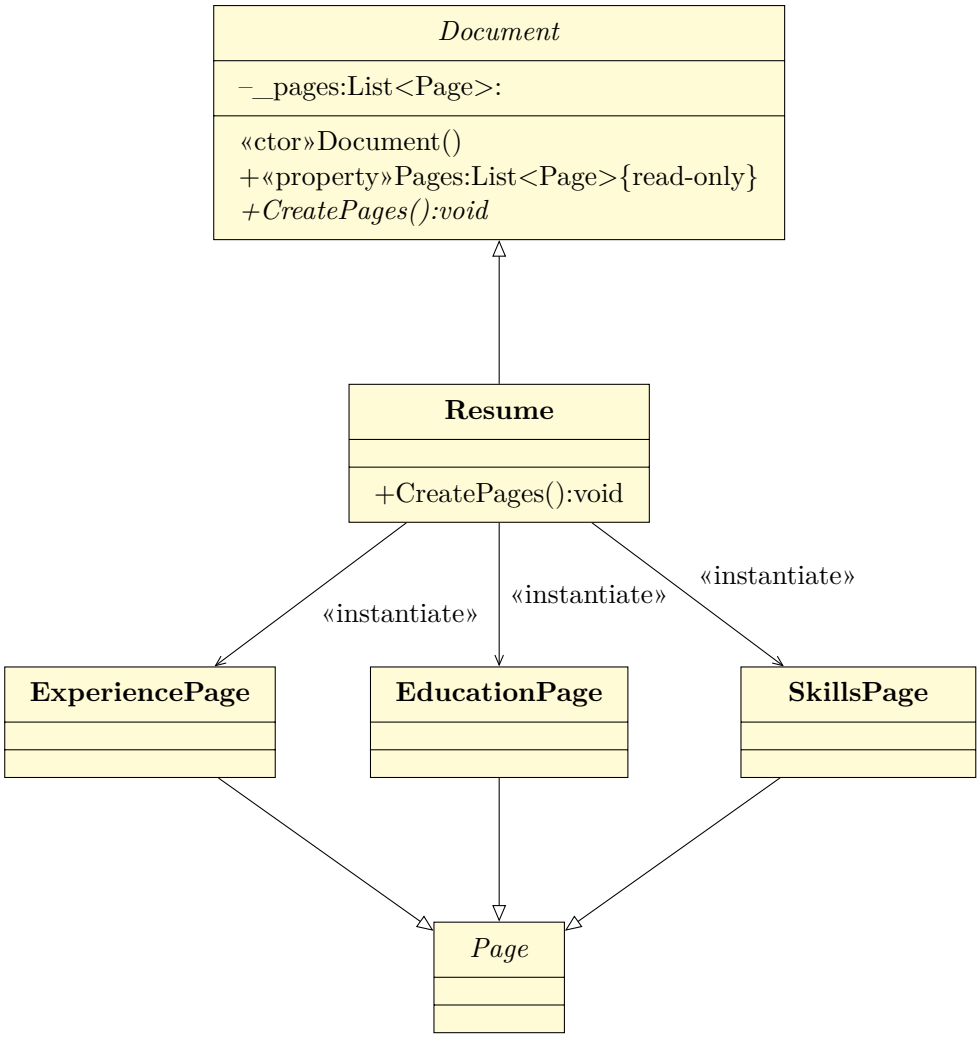


FIGURE 2 – Fabrique (*Factory Method*) : diagramme de classes générique

Explications :

— classe *Document*
  — la propriété de `Page` sera en lecteur seulement (*getter*)
  — le constructeur devra appeler la méthode fabrique de la classe `CreatePages()`
  — la méthode fabrique sera juste la déclaration d'une méthode abstraite [4]
— classe *Resume*
  — la classe heritera de la classe abstraite [5] de `Document`

4. classe abstraite : classe ne pouvant pas être instanciée.
5. classe abstraite : classe ne pouvant pas être instanciée.

— la classe aura la methode `CreatePages()` en ovveride
— classe `EducationPage`, `ExperiencePage` et `SkillsPage` : héritières de `Pages`
— classe `Page` : classe abstraite[6]

# 3 Références


— `http://www.dofactory.com/Patterns/Patterns.aspx`
— `https://fr.wikipedia.org/wiki/Fabrique_(patron_de_conception)`


6. classe abstraite : classe ne pouvant pas être instanciée.

## 3.1 Sources

```
12  /// <summary>
13  /// The 'Creator' abstract class
14  /// </summary>
15  abstract class Document
16  {
17      private List<Page> _pages = new List<Page>();
18
19      // Constructor calls abstract Factory method
20      public Document()
21      {
22          this.CreatePages();
23      }
24
25      public List<Page> Pages
26      {
27          get { return _pages; }
28      }
29
30      // Factory Method
31      public abstract void CreatePages();
32  }
```

Listing 1 – `Document.cs`

```
11  /// <summary>
12  /// A 'ConcreteCreator' class
13  /// </summary>
14  class Resume : Document
15  {
16      // Factory Method implementation
17      public override void CreatePages()
18      {
19          Pages.Add(new SkillsPage());
20
21          Pages.Add(new EducationPage());
22
23          Pages.Add(new ExperiencePage());
24      }
25  }
```

Listing 2 – `Resume.cs`

```
11  /// <summary>
12  /// The 'Product' abstract class
13  /// </summary>
14  abstract class Page
15  {
16
17  }
```

Listing 3 – `Page.cs`

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class SkillsPage : Page
15  {
16
17  }
```

Listing 4 – `SkillsPage.cs`

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class ExperiencePage : Page
15  {
16
17  }
```

Listing 5 – ExperiencePage.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class EducationPage : Page
15  {
16
17  }
```

Listing 6 – EducationPage.cs

```
11  /// <summary>
12  /// A 'ConcreteCreator' class
13  /// </summary>
14  class Report : Document
15  {
16      // Factory Method implementation
17      public override void CreatePages()
18      {
19          Pages.Add(new IntroductionPage());
20
21          Pages.Add(new ResultsPage());
22
23          Pages.Add(new ConclusionPage());
24
25          Pages.Add(new SummaryPage());
26
27          Pages.Add(new BibliographyPage());
28      }
29  }
```

Listing 7 – Report.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class IntroductionPage : Page
15  {
16
17  }
```

Listing 8 – IntroductionPage.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class SummaryPage : Page
15  {
16
17  }
```

Listing 9 – SummaryPage.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class ResultsPage : Page
15  {
16
17  }
```

Listing 10 – ResultsPage.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class ConclusionPage : Page
15  {
16
17  }
```

Listing 11 – ConclusionPage.cs

```
11  /// <summary>
12  /// A 'ConcreteProduct' class
13  /// </summary>
14  class BibliographyPage : Page
15  {
16
17  }
```

Listing 12 – BibliographyPage.cs

```
12  /// <summary>
13  /// Program startup class for Real-World
14  /// Factory Method Design Pattern.
15  /// </summary>
16  class Program
17  {
18      /// <summary>
19      /// Entry point into console application.
20      /// </summary>
21      static void Main(string[] args)
22      {
23          // Note: constructors call Factory Method
24          Document[] documents = new Document[2];
25
26          documents[0] = new Resume();
27          documents[1] = new Report();
28
29          // Display document pages
30          foreach (Document document in documents)
31          {
32              Console.WriteLine("\n" + document.GetType().Name + "--");
33
34              foreach (Page page in document.Pages)
35              {
36                  Console.WriteLine(" " + page.GetType().Name);
37              }
38          }
39
40          // Wait for user
41          Console.ReadKey();
42      }
43  }
```

Listing 13 – `Program.cs`

Console

```
Resume--
 SkillsPage
 EducationPage
 ExperiencePage

Report--
 IntroductionPage
 ResultsPage
 ConclusionPage
 SummaryPage
 BibliographyPage
```