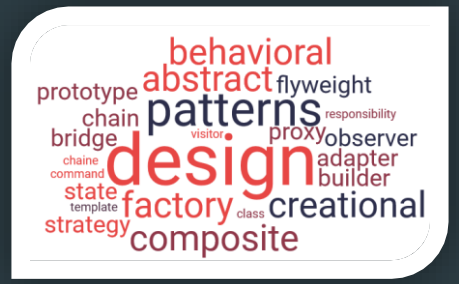


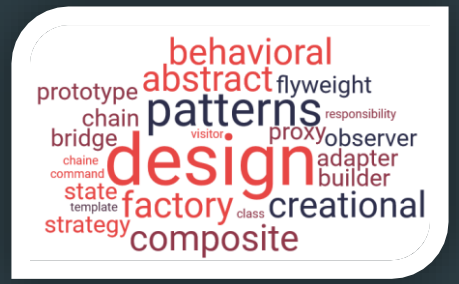
Les Patterns de construction

Introduction (1/2)

- Pour instancier un objet, il faut utiliser un constructeur
  - Il faut donc connaitre son type et disposer des informations nécessaires (paramètres du constructeur)
- Une telle approche crée un couplage fort entre les classes
  - Toute classe doit connaitre le type effectif de tout objet qu'il souhaite manipuler
  - Rend difficile l'évolution des classes
- Il est possible de déléguer la création des objets à une autre classe
  - Elle pourra se charger de l'utilisation du constructeur adéquat
  - Si nécessaire, elle pourra se charger de l'obtention des informations nécessaires à l'instanciation

Introduction (2/2)

- Les patterns de construction, de manière générale, abstraient le processus d'instanciation des objets
  - Permet à un client d'ignorer le type exact de l'objet utilisé
  - Permet d'écrire du code plus souple, puisque indépendant des classes concrètes
- Nous aborderons dans la suite deux patterns de construction
  - Factory Method (fabrique)
    - Introduit une méthode abstraite de création d'un objet dont le type dépend du contexte
  - Abstract Factory (fabrique abstraite)
    - Fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier



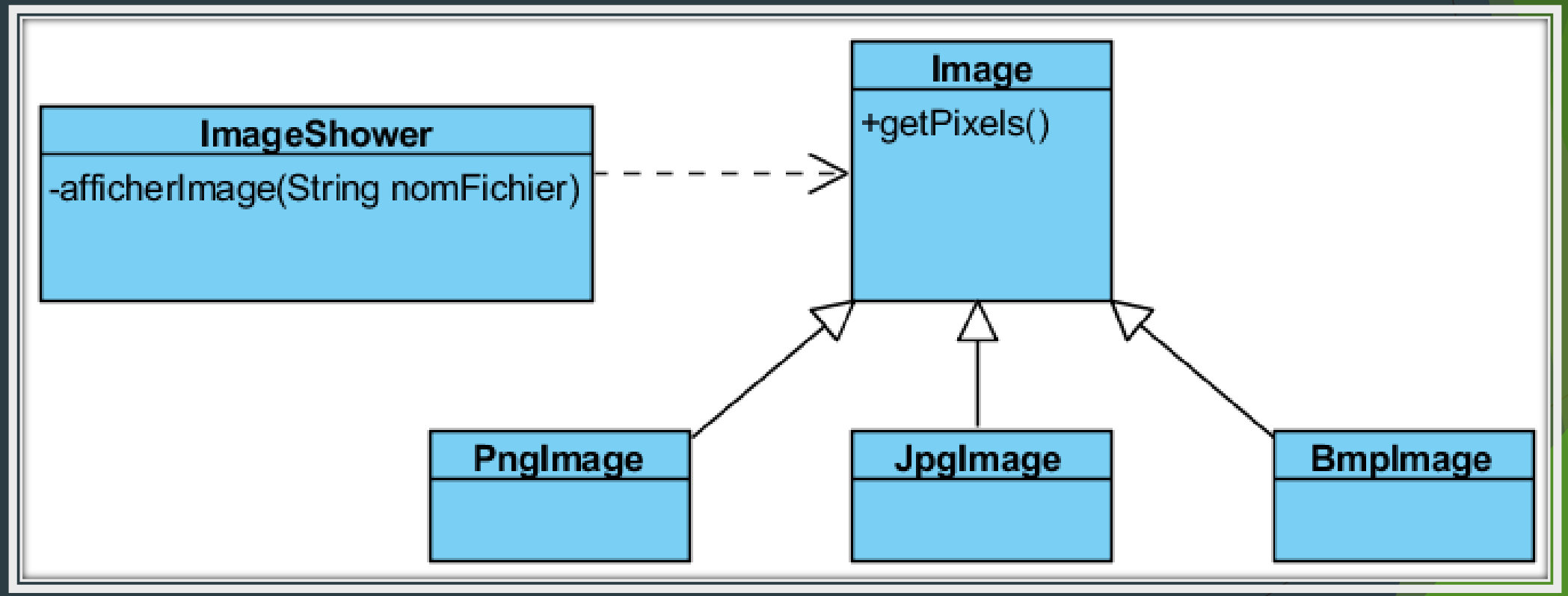
Le Pattern Factory Method (fabrique)

Cas illustratif

- Admettons que nous souhaitons mettre sur place une application du genre « visionneuse d'images »
  - Permet d'afficher des images
  - Gère plusieurs formats d'images (PNG, JPG, BMP, etc.)
- Pour ce faire, supposons que :
  - Nous disposons d'un ensemble de classes servant à manipuler des images et possédant toutes une méthode getPixels permettant de récupérer leurs pixels
  - Nous souhaitons pouvoir créer et utiliser une image, sans avoir besoin de connaitre sa classe exacte

Cas illustratif

- Avec une telle hierarchie de classe, ...



Cas illustratif

- Nous écrivons souvent du code ressemblant à ce fragment

```
public class ImageShower {
    ...
    public void afficherImage(String nomFicher) throws UnknownFormatException {
        Image image = null;
        int indexPoint = nomFichier.lastIndexOf(".");
        String extension = nomFichier.substring(++indexPoint);
        if (extension.equalsIgnoreCase('PNG')) {
            image = new PngImage(nomFicher);
        }
        else if (extension.equalsIgnoreCase('JPG')) {
            image = new JpgImage(nomFicher);
        }
        else if (extension.equalsIgnoreCase('BMP')) {
            image = new BmpImage(nomFicher);
        }
        else {
            throw new UnknownFormatException(extension);
        }
        byte pixels[][] = image.getPixels();
        this.showPixels(pixels);
    }
}
```

Cas illustratif : observations

- La décision de la classe à instancier est prise au niveau du client (ImageShower)
  - Le client est ainsi fortement dépendant des classes d'images
    - Toute modification au niveau des classes d'images peut affecter le client
  - Lorsqu'il faudra apporter des modifications ou des extensions, il faut reprendre ce code et examiner ce qu'il faudra ajouter ou supprimer
    - Si on souhaite gérer un nouveau format d'image, il faudra ajouter un autre cas de test partout où ce code était présent
  - Ce code choisissant la classe à instancier peut se trouver dans plusieurs parties de l'application
    - Rend ainsi la maintenance et les mises à jour plus difficiles et plus sujettes à des erreurs

Cas illustratif : observations

- Pourtant, le client a uniquement besoin des services offerts par les classes d'image, pas des classes en elles-mêmes
  - Il n'a besoin par exemple que d'une classe de lecture d'images PNG, pas de la classe PngImage en soi
    - N'importe quelle classe capable de récupérer les pixels d'une PNG pourrait remplacer PngImage
      - Que ce soit PngReader, OptimizedPngImage, ...
  - On peut aller plus loin pour dire que le client n'a besoin que d'une instance pouvant lui décoder l'image, peu importe sa classe
    - Il n'a pas besoin de connaitre l'intégralité des formats supportés
      - L'exception levée peut l'aider à savoir si le format est supporté ou non par exemple

Le Pattern Factory Method

- Quelques principes de conception
  - Soyez ouvert à l'extension, mais fermé à la modification
  - Encapsuler ce qui varie souvent
- Le processus d'instanciation de l'image est encapsulé dans une classe à part dénommée « fabrique »
  - Le client va désormais utiliser la fabrique pour obtenir des instances
    - Nul besoin pour le client de connaitre le type effectif de l'image
  - Ce qui est susceptible de varier est ainsi encapsulé dans la Fabrique
    - On pourra ainsi ajouter des formats d'images ou modifier des classes d'images existantes sans toucher au code du client

Le Pattern Factory Method

- Le client utilise désormais la Fabrique pour obtenir des instances d'images

```
public class ImageShower {
    ...
    public void afficherImage(String nomFicher) throws UnknownFormatException {
        Image image = FabriqueImage.getInstance(nomFicher);
        byte pixels[][] = image.getPixels();
        this.showPixels(pixels);
    }
}
```

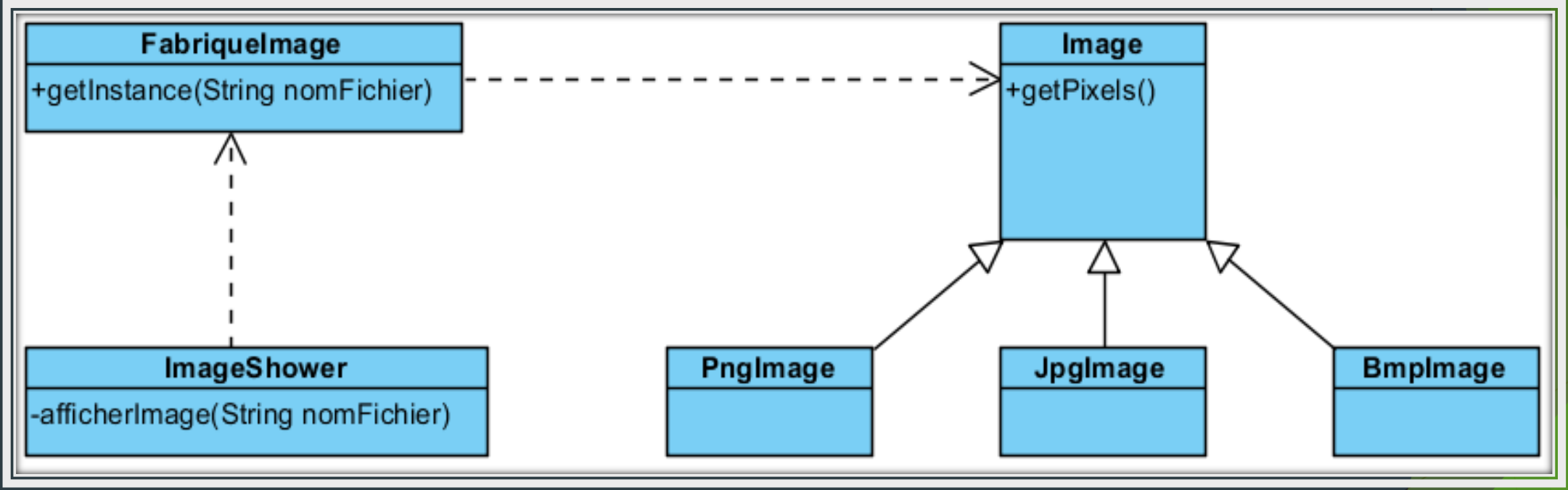
Le Pattern Factory Method

- Code de la fabrique d'images

```
class FabriqueImage {
    public static Image getInstance(String nomFicher) throws UnknownFormatException {
        Image image = null;
        int indexPoint = nomFichier.lastIndexOf(".");
        String extension = nomFichier.substring(++indexPoint);
        if (extension.equalsIgnoreCase('PNG')) {
            image = new PngImage(nomFicher);
        }
        else if (extension.equalsIgnoreCase('JPG')) {
            image = new JpgImage(nomFicher);
        }
        else if (extension.equalsIgnoreCase('BMP')) {
            image = new BmpImage(nomFicher);
        }
        else {
            throw new UnknownFormatException(extension);
        }
        return image;
    }
}
```

Le Pattern Factory Method

- La hierarchie des classes devient :

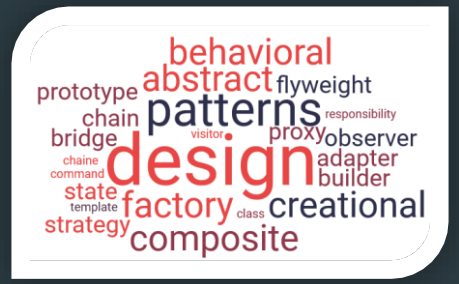


Le Pattern Factory Method

- ImageShower peut ainsi utiliser des images sans connaitre leurs types effectifs
  - Obtient des instances à travers la Fabrique
- Toute extension ou modification des classes d'images se fait au niveau de la Fabrique
- De telles évolutions du code n'impactent pas le client
  - On peut ajouter ou supprimer de nouveaux formats d'images sans que ImageShower n'aie besoin d'être modifiée
    - Code ouvert à l'extension mais fermé à la modification

Le Pattern Factory Method

- Il est possible avec ce pattern d'avoir plusieurs fabriques
  - On pourrait avoir plusieurs fabriques d'images en fonction des besoins
    - Une fabrique qui se base sur l'extension du nom du fichier pour déterminer la classe à choisir
      - Fournir un fichier JPG avec l'extension PNG pourrait faire bugger le programme (PngImage instanciée au lieu de JpgImage)
    - Une autre fabrique qui, en plus de l'extension, scrute le contenu du fichier pour choisir le type exact du fichier
      - Fournir un fichier JPG avec l'extension PNG ne devrait pas causer de problèmes au programme (JpgImage instanciée vu que la détection se fait suivant le contenu)



Le Pattern Abstract Factory (fabrique abstraite)