La désignation chainée (method chaining)

Adrien CAUBEL

20 septembre 2022

Table des matières

1 Le concept

1.1 Définition

La désignation chainée ou chainage de méthodes (*method chaining* en anglais) consiste à invoquer plusieurs méthodes sur un même objet en une seule instruction. Et ce dans le but d'améliorer la lisibilité du code.

1.2 Illustration

Pour mieux comprendre cette définition, regardons l'exemple suivant. Nous avons défini une classe `Maison` composée d'un ensemble de *setters*.

1.2.1 Sans désignation chainer

Chaque *setters* ne renvoyant aucun type (`void`) nous devons faire précéder les méthodes du nom de l'objet, en l'occurence `maison`.

Maison
- nbPortes : int - nbFenetres : int - hasJardin : boolean
+ setNbPortes(int nbPortes) + setNbFenetres(int nbFenetres) + setHasJardin(boolean hasJardin)

```
Maison maison = new Maison();
maison.setNbPortes(8);
maison.setNbFenetres(10);
maison.setHasJardin(true);
```

1.2.2 Avec la désignation chainée

La définition nous dit *d'agir en une seule instruction sur plusieurs méthodes du même objet*. En soit, au lieu de faire précéder le nom de la méthode par le même nom d'objet à chaque fois, ce nom d'objet reste actif tout au long de l'instruction. L'objectif est d'allèger la syntaxe en supprimant une information répétitive.

```
Maison maison = new Maison();
maison                          // on ne précise qu'une fois maison
.setNbPortes(8)
.setNbFenetres(10);
.setHasJardin(true);
```

2 Mise en place du chainage

2.1 Implémentation basique

La mise en place du chainage est simple. Pour chaque méthode de notre classe `Maison` qui modifie l'objet nous devons renvoyer ce même objet. Autrement dit, renvoyer `this`.

```
class Maison {
   private int nbPortes;
   private int nbFenetres;
   private boolean hasJardin;

   public Maison setNbPortes(int nbPortes) {
      this.nbPortes = nbPortes
      return this; // on renvoie l'objet appelant pour le chainage
   }

   public Maison setNbFenetres(int nbFenetres) {
      this.nbFenetres = nbFenetres
      return this; // on renvoie l'objet appelant pour le chainage
   }

   public Maison setNbFenetres(boolean hasJardin) {
      this.hasJardin = hasJardin
      return this; // on renvoie l'objet appelant pour le chainage
   }
}
```

```
Maison maison = new Maison();
maison                              // this == maison
.setNbPortes(8)                  // this est renvoyé
.setNbFenetres(10);              // this est renvoyé
.setHasJardin(true);             // this est renvoyé
```

2.2 Implémentation alternative avec un sous-objet

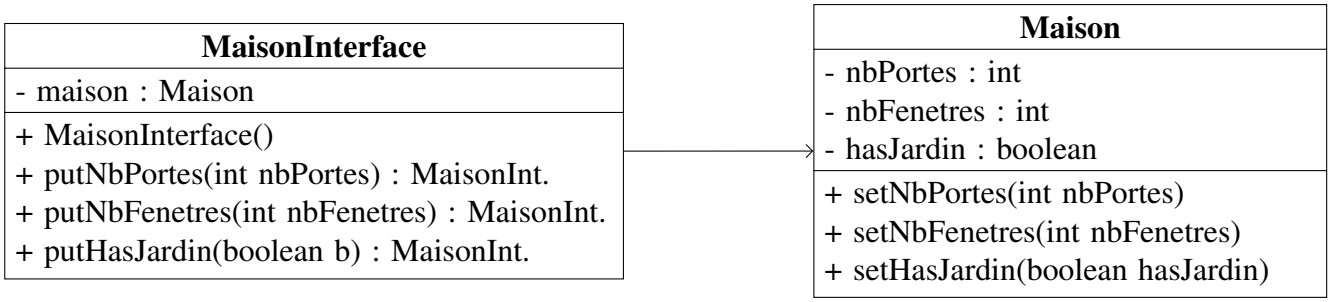
Cette alternative consiste à avoir un *sous-objet* qui dispose des méthodes de chainage (les méthodes `put`).

2.2.1 Définition primaire des classes

cette solution nécessite l'utilisation de deux classes :

— La première qui représente l'objet concret avec les *getters* et les *setters*.

— La seconde avec les méthodes qui vont modifier la première classe et retourner l'objet courant.



Les méthodes préfixées par `put` vont juste faire un appel aux *setters* de la classe `Maison` puis retourner l'objet courant. Par exemple pour la méthode `putNbPortes` nous obtenons le code suivant.

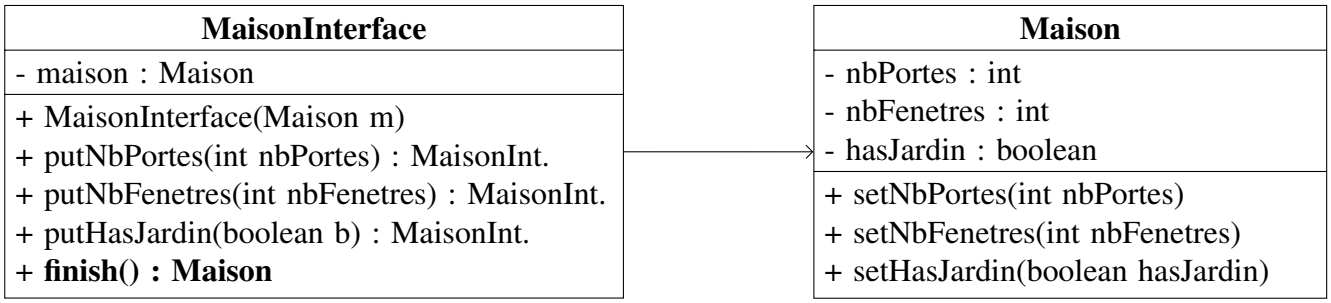
```
public MaisonInterface() {
   this.maison = new Maison(); // instanciation d'un objet maison
}

public MaisonInterface putNbPortes(int nb) {
   maison.setNbPortes(nb); // appel du setter de Maison
   return this;            // retourne l'objet de type MaisonInterface
}
```

Notre application fonctionne maintenant comme suit

```
MaisonBuilder mb = new MaisonBuilder();
mb.putNbPortes(10).putNbFenetre(12);
```

2.2.2 Terminer la définition des classes

Mais on ne peut pas s'arrêter à cette solution. En effet, ce que nous souhaitons c'est un objet de type `Maison`. Or pour le moment, nous n'avons qu'un objet typé `MaisonInterface`. Pour avoir le résultat souhaité, il nous suffit de rajouter une méthode qui retourne l'attribut `maison` dans la classe `MaisonInterface`.



```
MaisonBuilder mb = new MaisonBuilder();
Maison maison = mb.putNbPortes(10).putNbFenetre(12).finish();
```

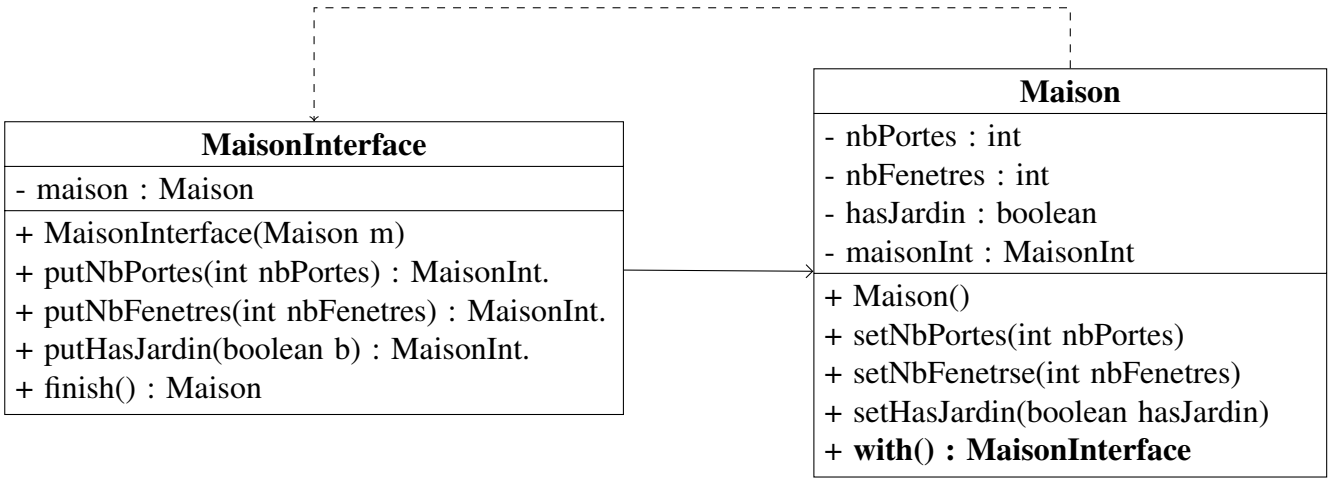
2.2.3 Cacher l'utilisation de la seconde classe (optionel)

On peut améliorer notre conception en masquant l'utilisation de `MaisonInterface`. En effet, pour le moment, nous devons :

- créer un objet `MaisonInterface`.
- appeler les méthodes préfixées par `put`.
- appeler la méthode `finish()` pour récupérer un objet de type `Maison`.

Mais on pourrait cacher l'utilisation de la classe `MaisonInterface` :

- Créer l'objet `MaisonInterface` dans la classe `Maison`. Via le constructeur
- Pouvoir dire qu'on va travailler avec l'objet `MaisonInterface`. Via la méthode `with`



Le constructeur `Maison` et la méthode `with()` sont définies comme suit

```
class Maison {
   public Maison() { maisonInterface = new MaisonInterface(this); }
   ...
   public MaisonInterface with() { return maisonInterface; }
}
```

Notre application cliente se définit maintenant avec le code suivant, où on vient effectivement de cacher l'utilisation de la classe `MaisonInterface`

```
main() {
   Maison maison =
   new Maison()      // instanciation cachée de MaisonInterface
   .with()           // renvoie maisonInterface
   .putNbPortes(10) // on peut donc utiliser ses méthodes
   .putNbFenetre(12)
   .finish();        // on renvoie l'objet maison
}
```

2.3 Conclusion sur ces implémentations

Ces solutions sont satisfaisantes si nous sommes dans une phase de conception de notre produit. Mais comment pouvons/nous rajouter le chainage sur une classe qui existe déjà et qui n'a pas été prévue pour ?

3 Faire du chainage dans une classe non chainable

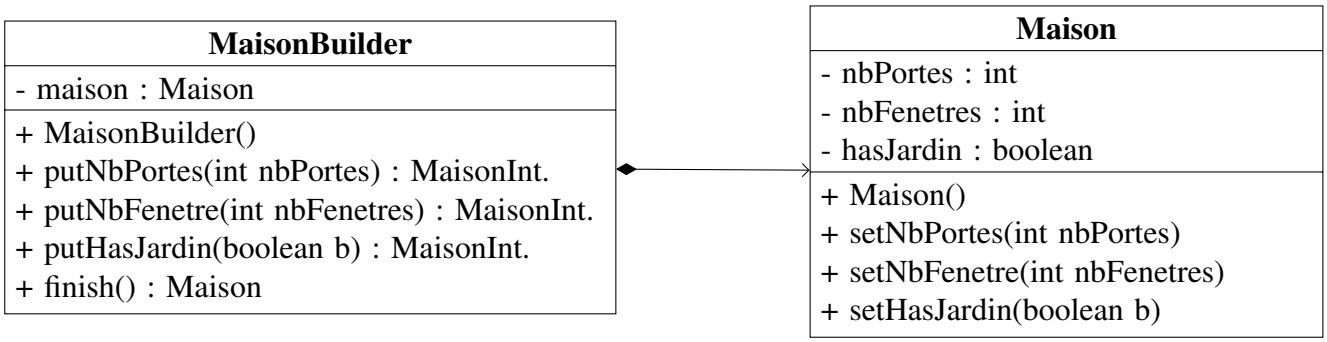
Il est fort possible que vous ayez déjà codé de nombreuses classes et que vous souhaitiez ajouter de la désignation chainée. Cependant, vous ne souhaitez pas toucher au code des classes existantes, car cela impliquerait de nombreux impacts sur le produit. Cependant nous pouvons arriver à notre bus en nous aidant du patron de conception *Builder*

3.1 Modification à apporter

Cette solution est semblable à celle développée au travers d'un sous-objet (section 2.2) mais quelques changements sont nécessaires :

- Supprimer l'attribut `maisonInterface` dans la classe `Maison`.
- Supprimer la méthode `with()` dans la classe `Maison`.
- Conserver l'initialisation de l'attribut `maison` dans le constructeur de la classe `MaisonBuilder` (précédemment `MaisonInterface`).

On reprend sous forme de diagramme UML les points précédents et un exemple de code client.




```
    public MaisonBuilder() {
       this.maison = new Maison();
    }

    public Maison finish() {
       return this.maison;
    }
```

```
    Maison m = new MaisonBuilder()
    .putNbPortes(10)
    .putNbFenetres(12)
    .finish();
}
```

3.2 Conclusion

Grâce à l'utilisation du patron *Builder* nous ne modifions pas le système de base. Nous apportons une nouvelle classe qui va permettre de faire la désignation chainée.

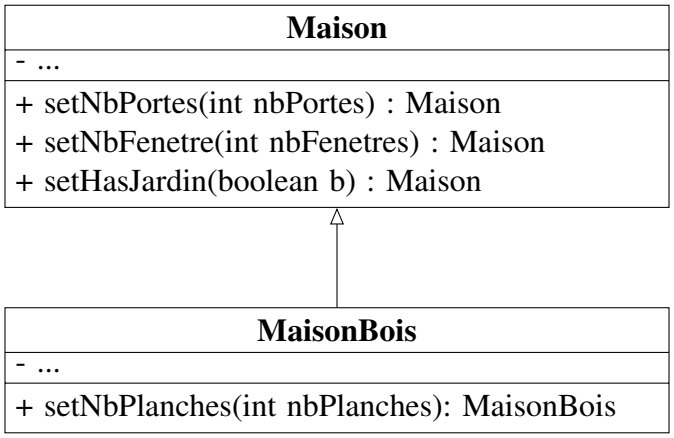
4 Les inconvénients de la désignation chainée

Les sections précédentes ont montré que la désignation chainée apporte une souplesse lors de l'écriture du code. Mais nous pouvons également nous demander quels sont les inconvénients d'une telle solution. En effet, le fait de pouvoir chainer les instructions et plus indirectement renvoyer l'objet courant ne pose-t-il pas un problème d'utilisation.

4.1 Un problème avec l'héritage

Pour illustrer le problème, nous n'utiliserons qu'une seule classe où nous modifierons les *setters* pour qu'ils renvoient l'objet appelant.

4.1.1 Présentation du problème



Nous souhaitons créer une maison en bois en utilisant la désignation chainée.

```
MaisonBois mb = new MaisonBois().setNbPortes(12).setNbPlanches(100);
```

On se rend compte que le code ne compile pas.

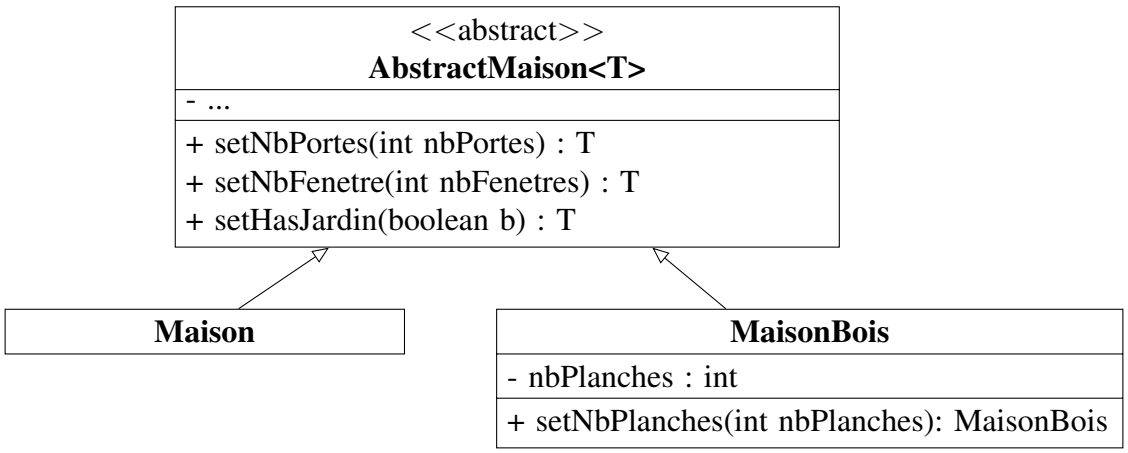
- La méthode `setNbPortes(12)` renvoie un objet de type `Maison`.
- Ensuite nous appelons `setNbPlanches(100)` mais cette méthode n'existe pour les maisons en bois
- Or, le compilateur ne sait pas qu'un objet de type `MaisonBois` est un sous-type de `Maison`.
- Donc il n'accepte pas l'instruction.

4.1.2 Une première solution

Cette première solution consiste à redéfinir toutes les méthodes de la super-classe dans la sous/classe, mais au lieu de renvoyer `Maison` nous devons renvoyer `MaisonBois`. Mais vous en conviendrez, on perd une partie du principe d'héritage.

4.1.3 Une seconde solution

Une seconde solution consisterait à faire en sorte que la méthode `setNbPortes()` me renvoie le bon type d'objet : le type concret. Pour ce faire, nous avons pouvons introduire le concept de généricité puis *caster* `this` lorsqu'on le renvoie à la fin de la méthode.




```
abstract class AbstractMaison<T> {
   public T setNbPortes(int nbPortes) {
      ...
      return (T)this; // on cast
   }
}
```

```
public class Maison extends AbstractMaison<Maison> {
   // T sera donc de type Maison
}
```

```
public class MaisonBois extends AbstractMaison<MaisonBois> {
   // T sera de type MaisonBois

   setNbPlanches(int nbPlanches) {
      this.nbPlanches = nbPlanches;
      return this
   }
}
```

Ainsi l'instruction précédente fonctionne, et on peut toujours créer des objets de type `Maison`.

```
MaisonBois mb = new MaisonBois().setNbPortes(12).setNbPlanches(100);

Maison m = new Maison().setNbPortes(12);
```

Notez que pour pouvoir créer des instances de la classe parente, nous avons dû la diviser en deux classes `AbstractMaison` et `Maison`, cette dernière est sans contenu.

Conclusion

Nous venons d'étudier le principe de désignation chainée. Il est assez simple à mettre en place lors de la conception d'une nouvelle application. Si l'application existe déjà et qu'il vous est impossible de la modifier des solutions existent comme l'utilisation du patron `Builder`. Ceci est quand même un peu contraignant, car vous allez coder une classe dans le simple but de pouvoir faire de la désignation chainée. Il faut donc s'assurer qu'elle sera essentielle dans la vie du logiciel.

Puis, nous avons terminé notre étude par un inconvénient majeur de la désignation chainée qu'est la mise en place de l'héritage.

Références

[1] Erich GAMMA, Richard HELM et Ralph JOHNSON. *Design patterns elements of reusable object oriented software*. Addison Wesley, 1998.

[2] Alexander SHVETS. *Design pattern Builder*. 2020. URL : `https://refactoring.guru/fr/design-patterns/builder`.