Un éditeur orienté ligne
Corrigé

Solution : Dans cette solution, des questions sont posées pour que vous puissiez continuer à faire les exercices même si vous avez commencé à lire la correction. Ce corrigé propose des indices. À vous de prendre un peu de temps pour les suivre avant de lire la suite du corrigé...

L'objectif de cet exercice est de construire un mini-éditeur orienté ligne qui respecte les commandes de `vi`. On veut qu'il propose un menu textuel donnant accès aux différentes commandes disponibles sur l'éditeur.

Pour simplifier, nous ferons les suppositions suivantes :

1. Nous nous limitons à l'édition d'une seule ligne.
2. L'interface utilisateur est minimale (figure 1). Elle contient :
   — le menu listant les opérations que peut réaliser l'utilisateur. L'utilisateur sélectionne une opération en tapant au clavier son numéro dans le menu.
   — la ligne en cours d'édition qui est affichée après chaque réalisation d'une opération par l'utilisateur. Le curseur est matérialisé par deux crochets encadrant le caractère courant. Dans le cas où la ligne est vide, le caractère tilde (~) est affiché et, bien sûr, le curseur n'est pas matérialisé.

```
Je suis la ligne [e]n cours d'édition !
------------------------------------------------------
 1) Ajouter un caractère au début de la ligne      [I]
 2) Ajouter un caractère à la fin de la ligne      [A]
 3) Placer le curseur au début de la ligne         [0]
 4) Avancer le curseur d'une position à droite     [l]
 5) Reculer le curseur d'une position à gauche     [h]
 6) Remplacer le caractère sous le curseur         [r]
 7) Supprimer le caractère sous le curseur         [x]
 8) Ajouter un caractère avant le curseur          [i]
 9) Ajouter un caractère après le curseur          [a]
10) Supprimer tous les caractères de la ligne     [dd]
 0) Quitter
------------------------------------------------------
Votre choix :
```

FIGURE 1 – Exemple de l'interface textuelle de l'éditeur

Il est reconnu que pour qu'une interface homme–machine soit conviviale, il faut que les opérations impossibles à réaliser ne puissent pas être sélectionnées par l'utilisateur. Par exemple, si le curseur est sur le dernier caractère de la ligne, il ne doit pas pouvoir sélectionner l'opération qui consiste à avancer le curseur. Généralement, les entrées des menus ne pouvant pas être exécutées sont grisées. Dans notre cas, nous supprimerons le numéro d'accès. La figure 2 montre

l'état de l'éditeur juste après son lancement sur une ligne vide. On constate, par l'absence de leur numéro d'accès, que de nombreuses opérations ne sont pas réalisables.

```
~
------------------------------------------------------
 1) Ajouter un caractère au début de la ligne      [I]
 2) Ajouter un caractère à la fin de la ligne      [A]
 -) Placer le curseur au début de la ligne         [0]
 -) Avancer le curseur d'une position à droite     [l]
 -) Reculer le curseur d'une position à gauche     [h]
 -) Remplacer le caractère sous le curseur         [r]
 -) Supprimer le caractère sous le curseur         [x]
 -) Ajouter un caractère avant le curseur          [i]
 -) Ajouter un caractère après le curseur          [a]
10) Supprimer tous les caractères de la ligne     [dd]
 0) Quitter
------------------------------------------------------
Votre choix :
```

FIGURE 2 – État de l'éditeur juste après son lancement (ligne vide)

Exercice 1 : Comprendre le problème posé
Résumer en une phrase l'application à développer.
Solution : Pas de solution donnée pour l'instant. C'est à vous de proposer votre phrase. On y reviendra à la fin...

Dans la suite, nous définissons la notion de ligne, puis nous proposons une modélisation de l'éditeur orienté ligne en définissant des menus textuels *réutilisables* qui prennent en compte la notion d'opérations non réalisables.

Remarque : Il n'est pas nécessaire de connaître `vi`. Les touches de commandes `vi` sont données à titre indicatif entre crochets. Elles ne seront pas exploitées dans la suite.
Attention : L'éditeur orienté ligne n'est qu'un prétexte. Les principes présentés ici pourraient être appliqués dans d'autres contextes.

Exercice 2 : Définition d'une ligne
La « spécification » de la ligne vous est donnée au listing 1. Une ligne contient un nombre quelconque de caractères. Il est possible de rajouter un caractère au début ou à la fin de la ligne. La ligne définit un curseur qui peut être déplacé vers la droite (avancer) ou vers la gauche (reculer). Le caractère sous le curseur est appelé caractère courant. Il est possible d'effectuer des opérations relatives au curseur (remplacer le caractère courant, le supprimer, insérer un caractère avant ou après le caractère courant).
2.1. On décide de stocker les caractères de la ligne dans un tableau et de représenter le curseur par un entier. Écrire une réalisation (concrétisation) `LigneTab` de l'interface `Ligne`. On écrira le début de la classe, les attributs, se limitant aux opérations listées ci-après : un constructeur qui prend en paramètre la capacité de la ligne et construit une ligne vide, `avancer` et `raz`.
Solution : Les principaux points sont les suivants.

1. Une réalisation est demandée. Il faut mettre `implements Ligne`. Les intérêts sont multiples (outre le fait que c'est demandé) :

— Le premier, c'est qu'ainsi une `LigneTab` pourra être utilisée partout où une `Ligne` est attendue (sous-typage, principe de substitution).
— Si on oublie d'implanter certaines méthodes de l'interface le compilateur signalera un message d'erreur en indiquant les méthodes qui sont abstraites (et doc oubliées).
— La documentation et les contrats de `Ligne` s'appliquent à `LigneTab`.

2. Le choix des attributs. Plusieurs choix seraient possibles mais comme on nous impose d'utiliser un tableau, on devrait en avoir 3 pour stocker :
   (a) les caractères de la ligne (un tableau de caractères). On peut donc l'appeler `caracteres` (et pas `tab` ni `ligne`)
   (b) la taille effective de ce tableau (un entier). On peut l'appeler taille (effective du tableau), longueur (de la ligne), etc.
   (c) le curseur (un entier).

   Notons qu'il n'est pas nécessaire de (et qu'il ne faut pas !) conserver la capacité du tableau car on peut la retrouver en faisant `caracteres.length`.

   On peut définir des **invariants de liaison** pour faire le lien entre la spécification (l'interface `Ligne`) et le choix d'implantation que l'on vient de faire dans `LigneTab`. Par exemple, on peut dire que `getLongueur()` correspond avec `this.longueur`, etc. Notons que ces invariants doivent être déclarés `private` car ils portent sur des éléments déclarés `private` (les attributs). Ils ne seront donc visibles que du programmeur de la classe.
   Bien noter que c'est l'attribut `curseur` qui modélise la position du curseur. Il n'y aura pas de ~ ou de crochets dans le tableau `caracteres`, c'est dans la méthode `afficher` que l'on écrira ces caractères. C'est la différence entre la représentation de l'information (comment on décide de la représenter/modéliser dans le programme) et sa présentation (comment en décide la présenter/restituer/afficher à l'utilisateur du programme).

```
1 public class LigneTab implements Ligne {
2     //@ // Invariants de liaison
3     //@ private invariant getLongueur() == longueur;
4     //@ private invariant getCurseur() == curseur;
5     //@ private invariant (\forall int i; 1 <= i && i <= longueur;
6     //@         ieme(i) == caracteres[i-1]);
7     private char[] caracteres;    // les caractères de la ligne
8     private int longueur;        // nombre de caractères de la ligne
9     private int curseur;        // position du curseur sur la ligne (1 au début)
```

Maintenant que l'on a fait le choix des attributs, on peut écrire le constructeur. À vous de le faire avant de lire la suite...

Voici le constructeur. Il prend en paramètre la capacité initiale de la ligne.

```
1       /** Initialiser la ligne à vide avec la capacité indiquée.
2        * @param capacite capacité initiale de la ligne
3        */
4       public LigneTab(int capacite) {
5           this.caracteres = new char[capacite];
6           this.curseur = this.longueur = 0;
7       }
```

Attention : Pour ce constructeur il faut écrire la spécification complète car elle n'est pas donnée dans l'interface.

Et maintenant on peut écrire les deux opérations demandées `avancer`, puis `raz`. À vous de jouer...

```
1       public void avancer() {
2           this.curseur++;
3       }
```

Ici, il est logique que le code se limite à cette seule instruction car on est dans un contexte de programmation par contrat (voir l'interface `Ligne` qui possède des contrats JML). Quand on écrit le code de `avancer`, on part du principe que la précondition est vraie (l'appelant l'a vérifiée, c'est son obligation et du coup le bénéfice pour la méthode appelée). La méthode doit établir la postcondition (c'est l'obligation de l'appelé et le bénéfice de l'appelant).

Si on n'était pas dans un contexte de programmation par contrat. Comment aurait-on fait ?

Il y a un problème si on est à la fin de ligne. Le réflexe est d'ajouter une condition...

```
1   public void avancer() {
2       if (this.curseur < this.longueur) {
3           this.curseur++;
4       }
5   }
```

Est-ce une bonne idée ?

Il faut toujours se poser la question du **`else`**. Dans le code précédent on a décidé de ne rien faire, et donc de laisser le curseur sur le même caractère, le dernier. En fait, plusieurs solutions sont possibles :

1. Rester sur le dernier caractère,
2. Aller sur le premier caractère de la ligne,
3. Ajouter un espace à droite et se placer le curseur sur cet espace,
4. Se placer sur le premier caractère de la ligne suivante.

Ceci devrait être alors spécifié dans la méthode `avancer` de l'interface `Ligne`. Mais est-ce logique de choisir une stratégie ici ? Est-ce qu'on pourra vraiment implanter la dernière stratégie ? Non, car sur une ligne on n'a pas accès à la ligne suivante !

En conclusion, il est certainement préférable de lever une exception ce qui permettra à l'appelant de choisir sa stratégie. Le code s'écrirait ainsi :

```
1   public void avancer() {
2       if (this.curseur >= this.longueur) {
3           throw new HorsLigneException();
4       }
5
6       this.curseur++;
7   }
```

Notez le traitement des exceptions en début de méthode. Il n'y a pas de **`else`** car le **`throw`** arrêtera l'exécution de la méthode. Ceci permet de garder le code « normal » bien visible, au premier niveau, pas caché dans un **`else`**.

Comment avez-vous écrit la méthode `raz`, abréviation de « remettre à zéro » ? Si vous ne l'avez pas fait, essayez de l'écrire maintenant avant de lire la suite...

Une solution qui est souvent donnée est la suivante.

```
1  public void raz() {
2      this.longueur = 0;
3      this.curseur = 0;
4  }
```

Bien sûr, ce n'est pas la bonne solution si on lit la documentation des cette méthode et ses contrats. En effet, il ne s'agit pas de remettre à zéro la ligne, mais le curseur de la ligne. Son nom est mal choisi. Il va donc induire une compréhension erronée de ce qu'elle fait et une mauvaise utilisation. **Conclusion : choisir avec soin les identifiants.**

Le bon code est le suivant.

```
1      public void raz() {
2          this.curseur = 1;
3      }
```

2.2. (facultative) Implanter les opérations suivantes : `remplacer`, `supprimer`, `ajouterFin`, et `afficher`.

Solution : Pas vraiment de difficulté, c'est juste algorithmique.

On constate un décalage entre la valeur du curseur et l'indice dans le tableau car le curseur sera toujours compris entre 1 et la longueur de la ligne quand la ligne est non vide. Il vaudra 0 quand la ligne est vide. C'est ce qui est exprimé par les invariants de l'interface.

TODO : ajouter le code correspondant... Pas nécessaire pour la suite.

Que se passe-t-il pour `ajouterFin` si le tableau `caracteres` est plein ?

Si le tableau est plein, il faut essayer de l'agrandir car dans la spécification il n'y a aucune précondition sur `ajouterFin`, on peut donc toujours l'appeler.

Bien sûr, quand on va allouer un nouveau tableau plus grand on risque d'avoir une erreur (`OutOfMemoryError`). Mais s'il n'y a plus de mémoire, ce sera difficile d'aller plus loin...

Exercice 3 : Réalisation de l'éditeur (et des menus)
Proposer (en 10 minutes maximum) une manière de programmer en Java l'éditeur orienté ligne.
Solution :

Pas d'idée pour démarrer... Et réussir à le faire dans le temps très court de 10 minutes ?
Indice : On peut utiliser UML pour décrire l'architecture du système et des annotations pour donner le pseudo-code des principales méthodes. Raffinages ! Raffinages ! C'est aussi un moyen de rester synthétique on donnant l'intention (l'objectif) et pas tous les détails. C'est aussi ce qui permettra d'identifier de nouvelles méthodes (les actions complexes).

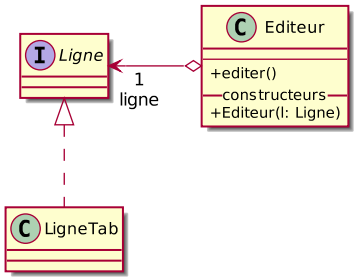
Essayez encore un peu...

On veut faire un éditeur (classe `Éditeur`) d'une ligne (interface `Ligne`). À quoi sert un éditeur ? À éditer. On aura donc certainement une méthodes `éditer`.

Quelle relation entre les `Éditeur` et `Ligne` ? Au moins une relation de dépendance (utilisation) : un éditeur permet d'éditer une ligne.

La ligne pourrait être en paramètre de la méthode `éditer`. Dans ce cas, on aurait juste une relation de dépendance. Dans la suite, on va considérer que `éditer` n'a pas de paramètre et que l'on donnera la ligne en paramètre du constructeur de `Éditeur`. On a donc une relation d'association, voire d'agrégation (il faut une ligne pour pouvoir utiliser l'éditeur). Pas composition car ce n'est pas l'éditeur qui est responsable de la ligne : elle existait avant de lancer l'éditeur et elle devrait continuer à exister après (sinon, pourquoi éditer une ligne ?).

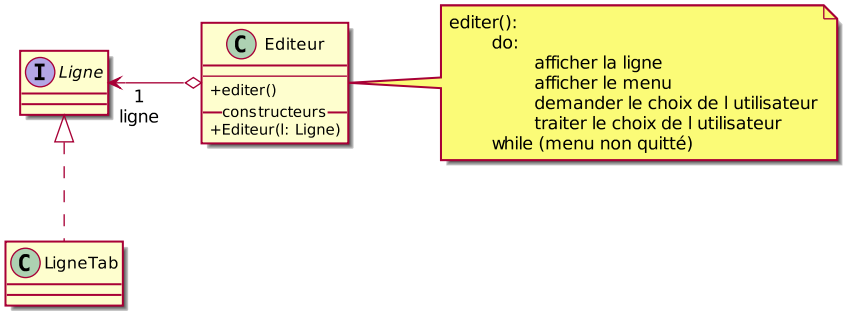
Inutile de détailler `Ligne` : ceci a été fait dans l'exercice précédent et on a écrit une réalisation `LigneTab`.



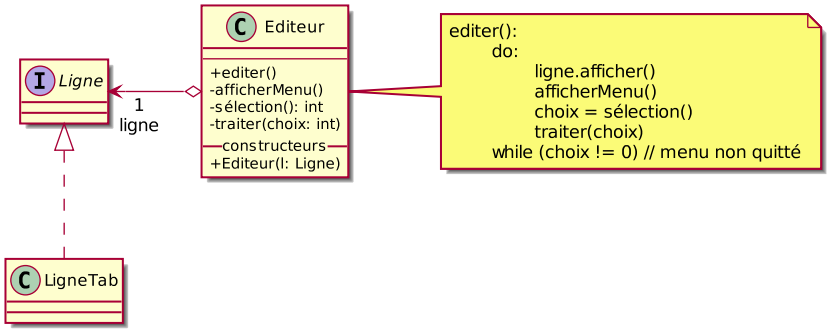
Il ne nous reste plus qu'à détailler `éditer`...

On peut le faire dans une petite annotation... À vous...

Voici le diagramme avec la méthode `éditer` détaillée.



On constate qu'elle contient plusieurs actions complexes qui vont naturellement devenir des méthodes privées. Pour "afficher la ligne", on a déjà la méthode abstraite correspondante sur l'interface `Ligne`. On va donc ajouter les méthodes `afficherMenu` (afficher le menu), `sélection` (demander le choix de l'utilisateur) et `traiter` (traiter le choix de l'utilisateur).



Il faut à leur tour les raffiner.

À vous...

Regardons comment faire...

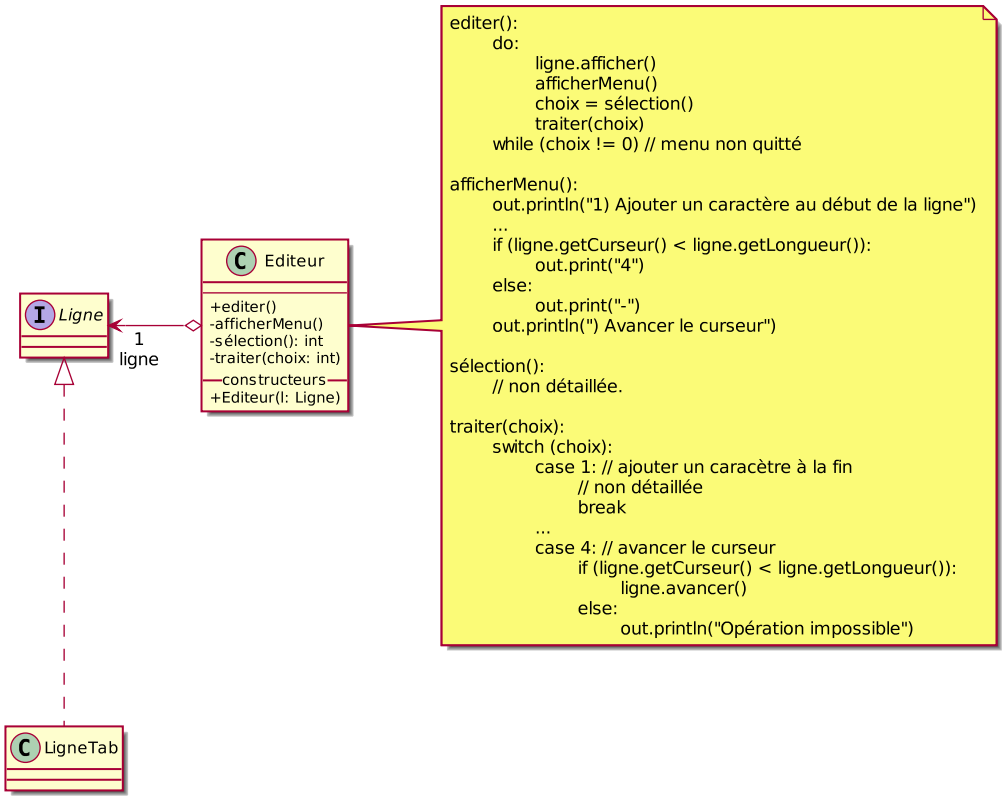
— Pour `afficherMenu`, il faut savoir si le numéro doit être affiché ou pas. Pour certaines opérations, il le sera toujours comme ajouter au début ou à la fin. Pour d'autres, il faudra le tester explicitement. La condition à tester coïncide généralement à la précondition de la méthode correspondante de ligne. Par exemple pour avancer le curseur, il faut :

```
ligne.getCurseur() < ligne.getLongueur()
```

— On considère que `sélection` se contente de demander un entier à l'utilisateur jusqu'à ce qu'il lui fournisse un entier compris entre 0 et 10. On aurait aussi pu vérifier que le numéro saisi correspond bien à une opération possible. Comme on ne l'a pas fait ici, il faudra que `traiter` le fasse. En fait, il faut bien définir qui fait quoi.

— `traiter` correspond à un traitement par cas. Pour chaque numéro d'opération, on doit vérifier si l'opération est possible ou pas avant de l'exécuter. Pour ajouter au début, c'est toujours possible. Pour avancer le curseur, il faut que (même condition que dans `afficher`) :

```
ligne.getCurseur() < ligne.getLongueur()
```



Listing 1 – La spécification d'une ligne

```
1  /** Spécification d'une ligne de texte.
2    * @author Xavier Crégut (cregut@enseeiht.fr)
3    */
4  public interface Ligne {
5        //@ public invariant 0 <= getLongueur(); // La longueur est positive
```

```
6       //@
7       //@ // Le curseur est toujours sur un caractère sauf si la ligne est vide.
8       //@ public invariant 0 <= getCurseur() && getCurseur() <= getLongueur();
9       //@ public invariant getCurseur() == 0 <==> getLongueur() == 0;
10
11      /** nombre de caractères dans la ligne */
12      /*@ pure @*/ int getLongueur();
13
14      /** Position du curseur sur la ligne */
15      /*@ pure @*/ int getCurseur();
16
17      /** le ième caractère de la ligne
18       * @param i l'indice du caractère
19       * @return le ième caractère de la ligne
20       */
21      //@ requires 1 <= i && i <= getLongueur(); // indice valide
22      /*@ pure @*/ char ieme(int i);
23
24      /** Le caractère sous le curseur
25       */
26      //@ requires getLongueur() > 0; // la ligne est non vide
27      /*@ pure @*/ char getCourant();
28
29      /** Avancer le curseur d'une position à droite. */
30      //@ requires getCurseur() < getLongueur();  // pas à la fin
31      //@ ensures getCurseur() == \old(getCurseur()) + 1; // curseur avancé
32      void avancer();
33
34      /** Avancer le curseur d'une position à gauche. */
35      //@ requires getCurseur() > 1;      // pas en début de ligne
36      //@ ensures getCurseur() == \old(getCurseur()) - 1; // curseur reculé
37      void reculer();
38
39      /** Placer le curseur sur le premier caractère. */
40      //@ requires getLongueur() > 0; // ligne non vide
41      //@ ensures getCurseur() == 1; // curseur sur la première position
42      void raz();
43
44      /** Remplacer le caractère sous le curseur par le caractère c. */
45      //@ requires getLongueur() > 0;
46      //@ ensures getCourant() == c;
47      void remplacer(char c);
48
49      /** Supprimer le caractère sous le curseur. La position du curseur reste
50       * inchangée.
51       */
52      //@ requires getLongueur() > 0;
53      //@ ensures getLongueur() == \old(getLongueur()) - 1; // un caractère ôté
54      //@ ensures getCurseur() == Math.min(\old(getCurseur()), getLongueur());
55      void supprimer();
56
57      /** Ajouter le caractère c avant le curseur.
58       * Le curseur reste sur le même caractère.
59       */
60      //@ requires getLongueur() > 0;  // curseur positionné
61      //@
62      //@ ensures getLongueur() == \old(getLongueur()) + 1; // un caractère ajouté
63      //@ ensures getCurseur() == \old(getCurseur()) + 1; // curseur inchangé
64      //@ ensures getCourant() == \old(getCourant());
65      void ajouterAvant(char c);
66
67      /** Ajouter le caractère c après le curseur.
68       * Le curseur reste sur le même caractère.
```

```
69          */
70          //@ requires getLongueur() > 0;  // curseur positionné
71          //@ ensures getLongueur() == \old(getLongueur()) + 1; // caractère ajouté
72          //@ ensures getCurseur() == \old(getCurseur()); // curseur inchangé
73          //@ ensures getCourant() == \old(getCourant());
74          void ajouterApres(char c);
75
76          /** Afficher la ligne en mettant entre crochets [] le caractère courant.
77           * Si la ligne est vide, un seul caractère tilde(~) est affiché.
78           */
79          /*@ pure @*/ void afficher();
80
81          /** Ajouter le caractère c à la fin de la ligne.
82           * Le curseur reste sur le même caractère.
83           */
84          //@ ensures getLongueur() == \old(getLongueur()) + 1; // caractère ajouté
85          //@ ensures ieme(getLongueur()) == c;   // à la fin
86          //@ ensures (\forall int i; 1 <= i && i <= \old(getLongueur());
87          //@                                  ieme(i) == \old(ieme(i)));
88          //@ ensures getLongueur() > 1 ==> getCourant() == \old(getCourant());
89          //@ ensures getCurseur() == Math.max(1, \old(getCurseur()));
90          void ajouterFin(char c);
91
92          /** Ajouter le caractère c au début de la ligne
93           * Le curseur reste sur le même caractère.
94           */
95          //@ ensures getLongueur() == \old(getLongueur()) + 1; // caractère ajouté
96          //@ ensures ieme(1) == c;                 // en première position
97          //@ ensures (\forall int j; j >= 2 && j <= getLongueur();
98          //@                              ieme((int)j) == \old(ieme((int)(j-1))));
99          //@ ensures getLongueur() > 1 ==> getCourant() == \old(getCourant());
100         //@ ensures getCurseur() == \old(getCurseur()) + 1;
101         void ajouterDebut(char c);
102
103         /** supprimer le premier caractère de la ligne. Le curseur reste sur le
104          * même caractère.
105          */
106         //@ requires getLongueur() > 0;
107         //@ ensures getLongueur() == \old(getLongueur()) - 1;
108         //@ ensures \old(getCurseur()) != 1 ==> getCourant() == \old(getCourant());
109         //@ ensures getCurseur()
110         //@         == Math.min(Math.max((int)(\old(getCurseur())-1), 1), getLongueur());
111         void supprimerPremier();
112
113         /** supprimer le dernier caractère de la ligne. Le curseur reste sur le même
114           * caractère.
115           */
116         //@ requires getLongueur() > 0;
117         //@ ensures getLongueur() == \old(getLongueur()) - 1;
118         //@ ensures \old(getCurseur()) < \old(getLongueur())
119         //@                       ==> getCourant() == \old(getCourant());
120         //@ ensures getCurseur() == Math.min(\old(getCurseur()), getLongueur());
121         void supprimerDernier();
122
123 }
```

Exercice 4 : Évaluation de l'éditeur et de ses menus

Pour évaluer l'application écrite dans l'exercice 3, nous allons envisager plusieurs extensions et/ou modifications qui pourraient être demandées par notre client (celui qui nous a demandé de développer cette application).

4.1. *Ajouter de nouvelles opérations.* Comment faire pour ajouter une nouvelle opération sur l'éditeur (et donc une nouvelle entrée dans le menu) ?

Solution : Pour ajouter une nouvelle opération, il faut :

— Modifier `afficherMenu` pour afficher le texte de cette nouvelle opération et afficher son numéro si elle est possible.

— Modifier la borne supérieure des entiers dans `sélection`.

— Modifier la méthode `traiter` pour traiter la nouvelle opération.

Ce n'est pas difficile à faire mais il faut intervenir à plusieurs endroits de l'application.

4.2. *Réorganisation des entrées du menu.* Comment modifier l'ordre des entrées dans le menu ?

Solution : Il faut :

— Permuter les lignes de la méthode `afficherMenu` et renuméroter,

— Répercuter les modifications de numéro dans traiter.

Pas difficile à faire mais fastidieux et source d'erreurs !

4.3. *Réutilisation du menu.* Que peut-on réutiliser concernant les menus si on doit développer une autre application qui utilise également des menus ?

Solution : En terme de code rien ou presque. On pourrait réutiliser `sélection` en changeant la borne supérieure.

La seule chose que l'on peut réutiliser c'est le principe (le raffinage de `editer` de la classe `Éditeur`) mais il faudra écrire tout le code !

4.4. *Aide sur les opérations.* Lorsque l'utilisateur tape un numéro d'accès correspondant à une opération non réalisable, nous lui indiquons qu'il ne peut pas choisir cette opération. Il serait plus agréable pour l'utilisateur de savoir pourquoi l'opération n'est pas réalisable. Comment modifier la gestion des menus pour que cette explication puisse être donnée à l'utilisateur ?

De la même manière, quand l'utilisateur sélectionne une entrée du menu, il serait possible de lui afficher une « bulle d'aide » qui lui explique brièvement ce que fait l'opération associée.

Solution : À l'image d'un menu graphique, quand on passe la souris sur une entrée du menu, on pourrait avoir une bulle d'aide qui explique à quoi correspond cette opération. Ceci pourrait aussi être fait avec notre menu textuel. On pourrait imaginer taper « aide 2 » pour avoir l'aide de la 2ème opération du menu.

Il faudrait donc ajouter une nouvelle méthode `aide` qui retournerait une chaîne de caractères en fonction du numéro de l'entrée. Donc un nouveau traitement par cas et un nouvel endroit à modifier si on ajoute une opération ou si on change leur ordre.

4.5. *Raccourcis clavier.* Comment faire pour définir un raccourci clavier pour accéder aux opérations (en plus des numéros) ? Ce sont les lettres entre crochets sur les figures 1 et 2.

Solution : Une solution consiste à modifier la méthode `sélection` pour qu'elle fasse la correspondance entre le raccourci et le numéro correspondant. Ce n'est pas très difficile à faire mais le refaire pour tous les menus sera pénible...

4.6. *Structuration du menu en sous-menus.* On constate que le menu est trop grand et on souhaite regrouper les opérations par thème avec, par exemple, un menu spécifique pour les opérations liées au curseur.

Le menu proposé ne comporte qu'une petite partie des opérations de l'éditeur et il est déjà long. Il serait donc souhaitable de pouvoir regrouper les opérations par thème et de les structurer

en sous-menus, par exemple en regroupant dans un sous-menu les opérations relatives au curseur. L'application propose alors un menu principal et des sous-menus.

Comment faire pour intégrer cette notion de menus et sous-menus dans notre éditeur ?

Remarque : Il existe deux types de sous-menu : les sous-menus qui ne permettent la sélection que d'une seule opération et qui disparaissent et les sous-menus qui restent affichés jusqu'à ce qu'ils soient quittés explicitement par l'utilisateur. Ces derniers sous-menus permettent de sélectionner plusieurs opérations.

Solution : Ça peut paraître difficile... mais ça ne l'est pas !

On va réutiliser le principe mis en œuvre : trois méthodes (`afficherMenu`, `sélection` et `traiter`) pour chaque menu et chaque sous-menu. Et dans le traiter d'un menu, il faudra utiliser les trois méthodes du sous-menu à déclencher...

Pas difficile mais vraiment fastidieux et source d'erreur !

4.7. *Pouvoir annuler une opération.* En plus de griser les opérations non réalisables, un autre aspect important pour une interface homme/système est la possibilité d'annuler la ou les dernières opérations réalisées. Ceci permet à l'utilisateur de faire des essais sans risque. L'éditeur devra permettre de défaire (et refaire) les dernières opérations exécutées.

Expliquer comment il serait possible de modifier l'éditeur pour autoriser l'annulation des opérations réalisées.

Solution : Avec la solution actuelle, ce sera difficile à faire...

4.8. *Modification dynamique des entrées du menu.* Comment faire pour ajouter ou supprimer des entrées du menu pendant l'exécution d'un programme ?

Remarque : Ceci peut être contradictoire avec les aspects ergonomie. En effet, si la structure du menu change, l'utilisateur peut ne plus s'y retrouver. Dans ce cas, il est préférable de griser l'entrée plutôt que de la supprimer. Construire dynamiquement un menu est en revanche utile pour gérer la liste des derniers fichiers édités, par exemple.

Solution : On ne peut pas. Le nombre d'entrées est codé en dur dans `afficherMenu`, `sélection` et `traiter` !

Conclusion : La version que l'on vient d'écrire est facile à comprendre, facile à mettre en œuvre mais elle est peu évolutive (toute modification obligera à modifier plusieurs parties de l'application en risquant de faire des erreurs) et non réutilisable (on pourra réutiliser le principe mais pas le code).

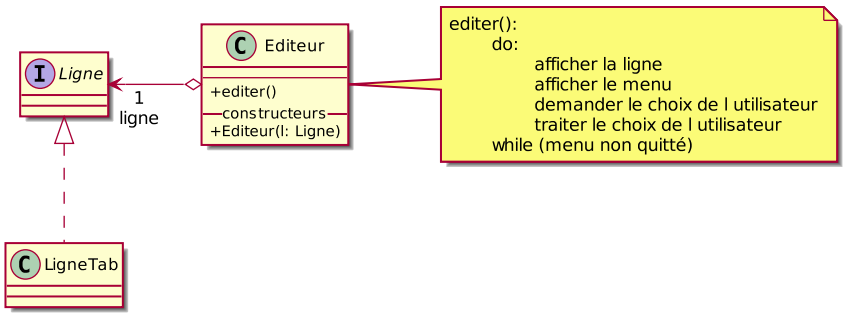
Les « ... » que nous avons utilisés montrent bien que notre solution a de mauvaises propriétés : trop de choses similaires, donc une mauvaise identification des concepts clés.

Fort de ces constatations, on reprend donc l'exercice 3 pour proposer une meilleure solution !

Exercice 5 : Réalisation de l'éditeur (et des menus)

Proposer (en 10 minutes maximum) une manière de programmer en Java l'éditeur orienté ligne. Cette fois-ci on veut qu'il soit *réutilisable*.

Solution : Le point de départ est le même. C'est toujours un éditeur d'une ligne. Et le principe de la méthode `éditer` est le même que tout à l'heure (c'était la partie réutilisable). On part donc à nouveau du diagramme suivant.



On veut que les menus soient réutilisables. Comment faire ?

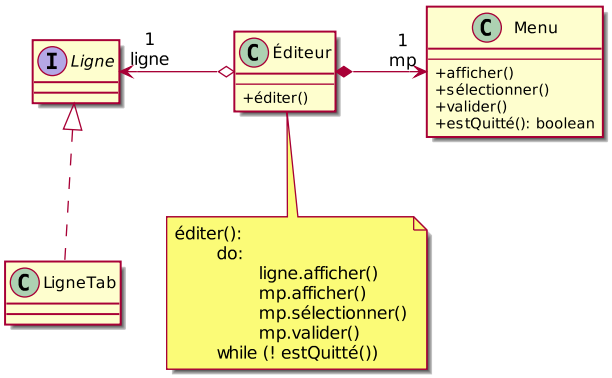
Il faut en faire une classe, la classe `Menu`. Il suffira alors de créer des objets de cette classe pour avoir autant de menus que souhaité et utiliser les méthodes fournies par la classe `Menu`.

1. Quelle est la relation entre `Éditeur` et `Menu` ?
2. Quelles sont les méthodes de `Menu` ?

C'est une relation de composition car le menu est propre à l'éditeur. Le menu constitue le menu principal de l'éditeur, on pourrait donc prendre comme rôle `menuPrincipal`. Comme ce nom est long, sur le diagramme on utilisera l'abréviation `mp`.

On aurait pu arriver à la même conclusion en raisonnant « objet » sur la base du raffinage de la méthode `éditer`. En effet « afficher une ligne » correspond à la méthode `afficher` de `Ligne`. De la même façon, « afficher le menu » suggère qu'il y a une classe `Menu` qui propose une méthode `afficher`. Ainsi, on aurait aussi une méthode `sélectionner` sur `Menu` (elle consiste à choisir un numéro dans le menu) et une méthode `valider` (l'équivalent de `traiter`). Par analogie avec un menu graphique, sélectionner est l'équivalent de déplacer une souris sur le menu et valider est l'équivalent de cliquer sur l'entrée.

Ces méthodes sont bien sûr des méthodes publiques de `Menu`.



Il s'agit maintenant de les détailler. On peut le faire dans n'importe quel ordre. Commençons par `afficher` que l'on va traiter complètement avant de détailler les autres.

Comment écrire la méthode `afficher` de `Menu` ? On pourra commencer par dire en français ce qu'elle fait pour en déduire le premier niveau de raffinage.

À vous...

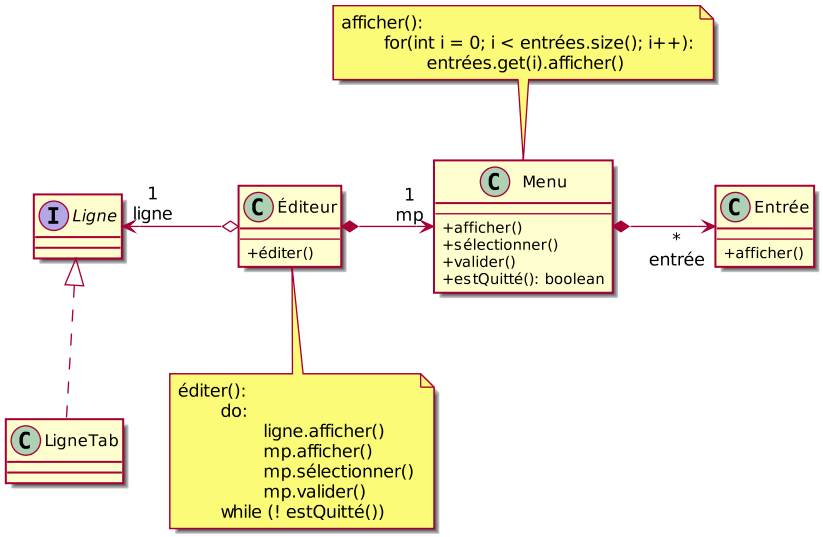
Afficher un menu, c'est afficher chacune de ses lignes. Le terme ligne est ici ambigu car il y a les lignes de l'éditeur et les lignes du menu. Choisissons un autre terme souvent utilisé pour les menus : entrée.

Ainsi, afficher un menu, c'est afficher chacune de ces entrées.

1. Comment écrire le raffinage correspondant ?
2. Que peut-on en déduire sur le diagramme de classe ?

Il s'agit donc d'afficher une entrée pour chaque entrée du menu. On peut en déduire qu'il existe une notion de `Entrée`. Et sur cette classe `Entrée`, il a une méthode afficher(). La classe `Entrée` est reliée à la classe `Menu` par une relation de composition : les entrées sont propres à un menu. Il peut y avoir plusieurs entrées pour un menu. On peut maintenant compléter le diagramme de classe.

Il faudrait aussi se demander comment traduire en Java la relation entre `Menu` et `Entrée`. On peut choisir de prendre une liste, `java.util.List`, d'entrées. On peut alors en déduire le code de la méthode `afficher` de `Menu`.



Remarque : On aurait pu utiliser un `foreach` au lieu d'un **`for`** avec indice explicite. On verra tout à l'heure pourquoi on ne l'a pas fait.

La méthode `Menu.afficher` est écrite mais elle dépend de `Entrée.afficher`.

Comment écrire la méthode `afficher` de `Entrée` ? On peut procéder comme tout à l'heure en répondant aux questions suivantes :

1. Quel est son objectif ?
2. Quel principe ?
3. Comment l'exprimer sous forme d'un raffinage ?

À vous...

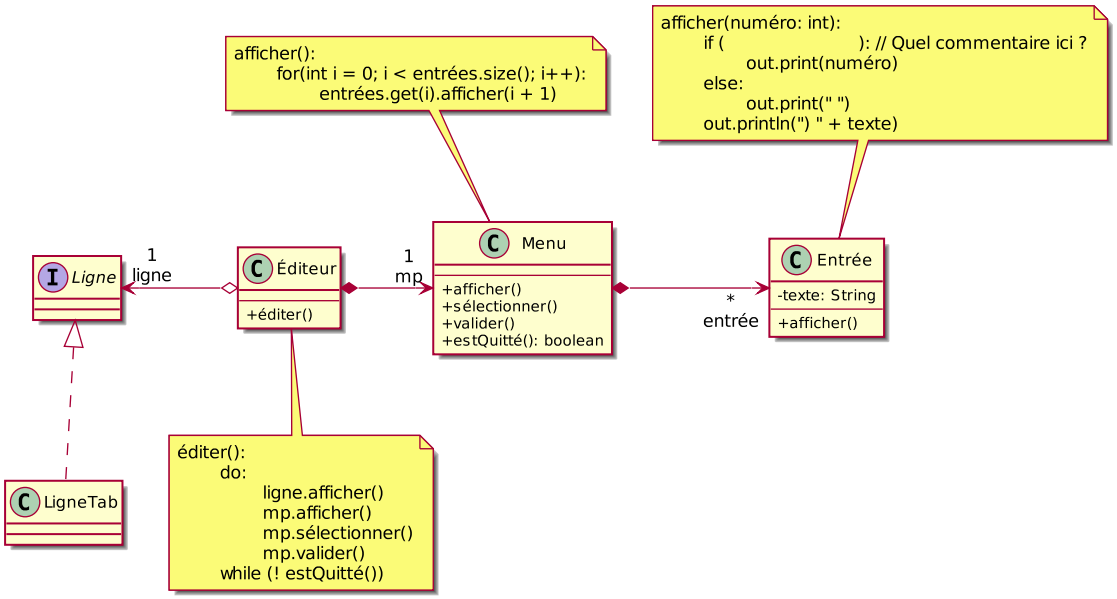
Principe : Il faut afficher le texte (par exemple « avancer le curseur ») mais avant il faut afficher le numéro... ou le griser.

La structure du raffinage est donc la suivante.

```
Comment « Entrée.afficher() » ?
    if (               ) {    // quel commentaire ici ?
        System.out.print(numero);
    } else {
        System.out.print('-');
    }
    System.out.println(texte)
```

Où est déclaré le texte ?

C’est certainement un attribut de la classe `Entrée`.



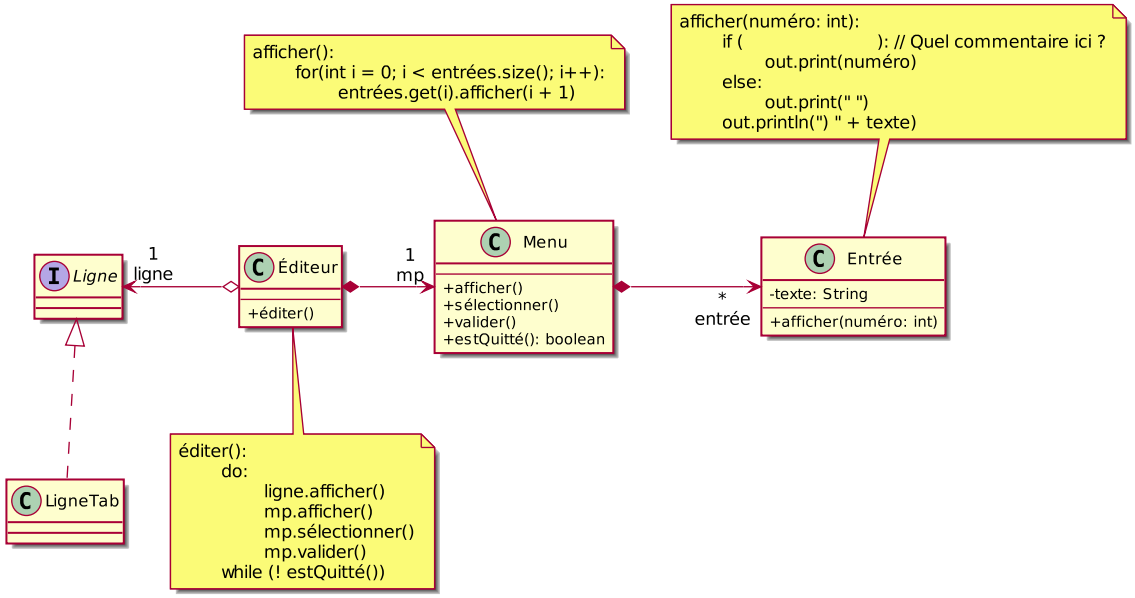
Où est déclaré le numéro ?

Le numéro pourrait aussi être défini comme un attribut de la classe `Entrée` mais le numéro dépend de la position de l'entrée dans la liste des entrées du `Menu`. Du coup, c'est plutôt le `Menu` qui connait le numéro. Où est donc déclaré le numéro ?

C'est donc un paramètre de la méthode afficher de `Entrée` et c'est la méthode `Menu.afficher` qui fournira le numéro que la méthode `Entrée.afficher` doit utiliser.

Quand une méthode doit accéder à une information, ce peut être un paramètre (le cas de numéro) ou un attribut de la classe (le cas de texte).

On arrive alors au diagramme de classe suivant.

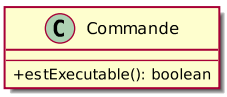


Il nous reste encore un point à voir. Quel est le commentaire à mettre après le **`if`**. En d'autres termes, à quelle condition affiche-t-on le numéro ? On pourra alors compléter le diagramme de classe et écrire la condition dans les parenthèses du **`if`**.

Indice : On affiche le numéro si l'opération associée à l'entrée est réalisable.

1. Quelle est la condition du `if` ?
2. Qu'en déduit-on sur le diagramme de classe ?

On a donc une nouvelle classe `Opération` qui possède une méthode `estRéalisable`. En fait, on va changer un peu les noms. On va utiliser `Commande` au lieu de `Opération` et `estExécutable` au lieu de `estRéalisable`. Ceci mettra mettre en évidence le patron de conception que nous allons mettre en œuvre dans ce TD : le patron « commande ».

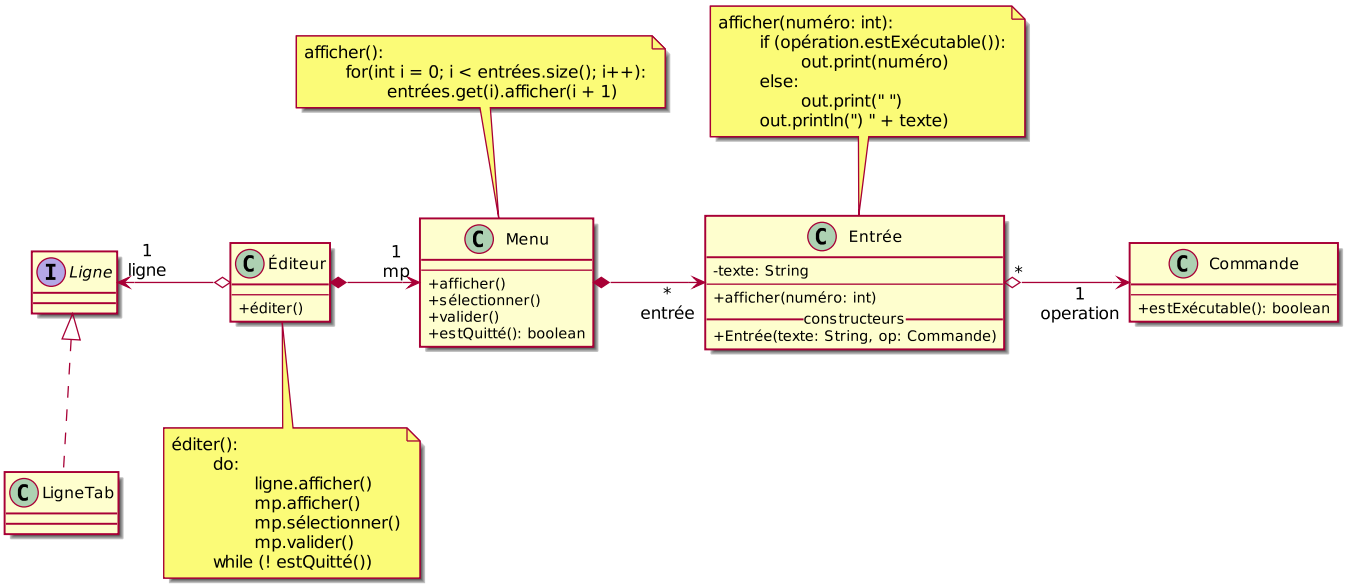


Le terme « opération » n'est pas perdu. Ce sera le rôle de `Commande` vis à vis de `Entrée`.

1. Quelle est la relation entre `Entrée` et `Commande` ?
2. Quelles multiplicités ?

C'est une relation d'agrégation car la même commande (opération) pourrait être partagée entre plusieurs menu (voire avec un bouton dans le cas d'une interface graphique).

Notre diagramme de classe progresse...



Remarque : On constate que le code de la méthode `Entrée.afficher`a la même structure de code que ce qu'on a fait pour afficher une entrée dans la méthode `afficherMenu` (première version de l'application). On a juste paramétré par le texte, le numéro et l'opération (la commande).

Il nous reste maintenant à écrire la méthode `estExécutable` de `Commande`. Comment fait-on ?

Indice : Si c'est l'opération « ajouter un caractère au début », on sait que c'est toujours possible. Le résultat de la méthode sera vrai.

Si c'est l'opération « avancer le curseur », le résultat sera

```
ligne.getCurseur() < ligne.getLongueur()
```

Si c'est une autre opération, le résultat sera certainement différent.

Que peut-on en conclure sur la méthode `Commande.estExécutable` ?

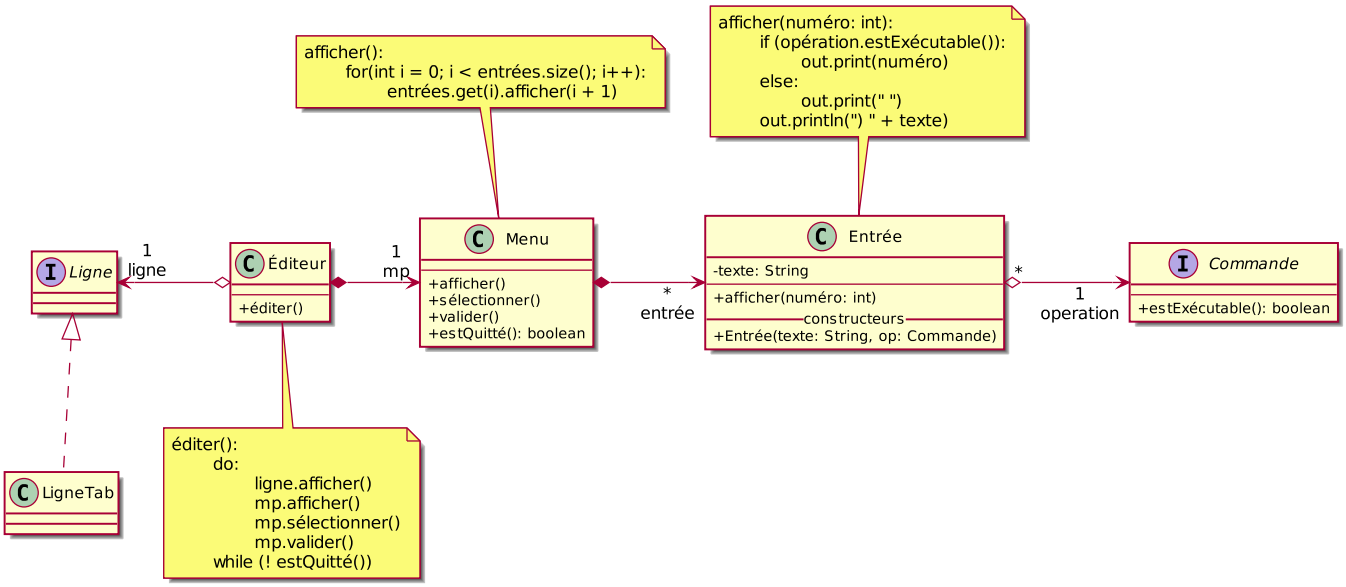
On en déduit qu'on sait l'écrire quand on connait l'opération. On constate qu'elle traduit les conditions d'application de la méthode de `Ligne` qui correspond à cette opération. Elle coïncide donc avec les conditions exprimées avec la programmation par contrat.

Mais on écrit la méthode `estExécutable` de `Commande`, on ne sait pas de quelle opération il s'agit. Comment faire ?

Et bien on sait que la méthode `estExécutable` existe : on veut savoir si une commande est exécutable ou non (i.e. si une opération est réalisable ou non). On en a besoin pour écrire le code `Entrée.afficher`. Cependant on se fait pas ce qu'elle fait tant qu'on ne connait pas l'opération précise. C'est donc...

une **méthode abstraite !**

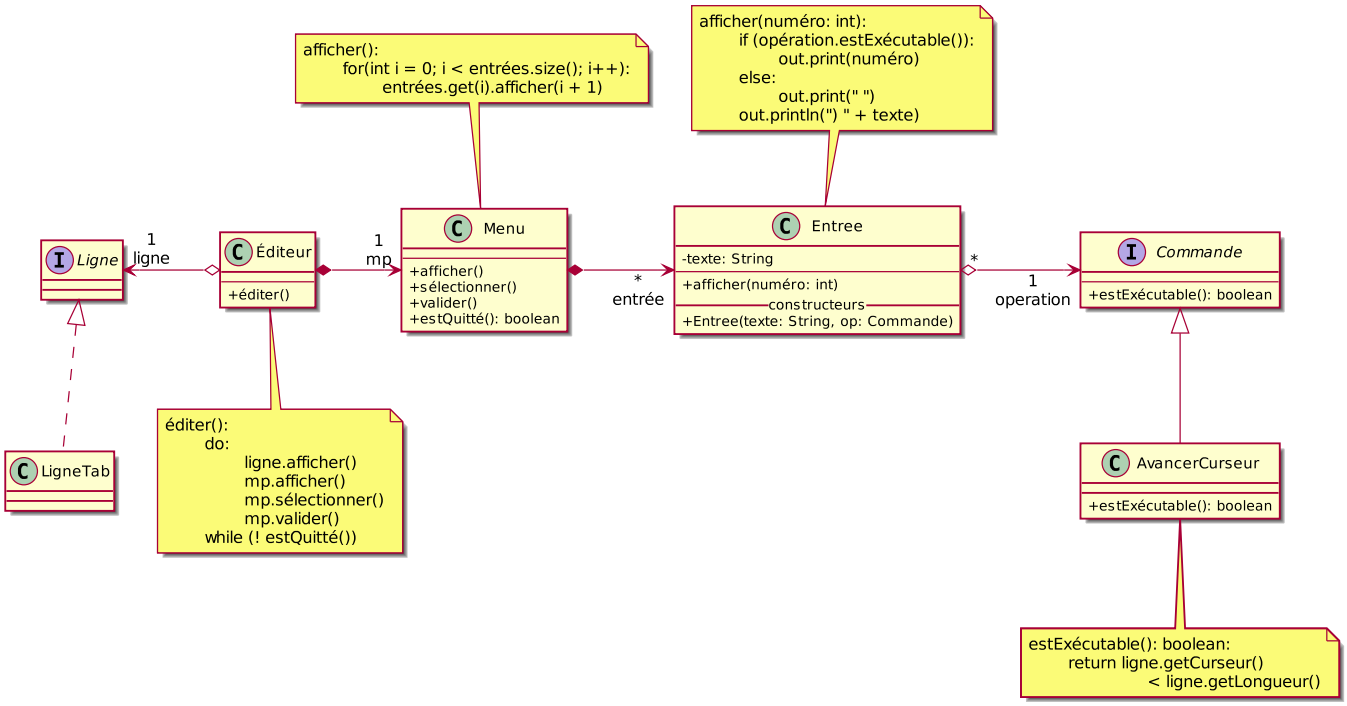
Et donc `Commande` est soit une classe abstraite soit une interface. Ici nous allons en faire une interface (rien à factoriser entre les sous-types).



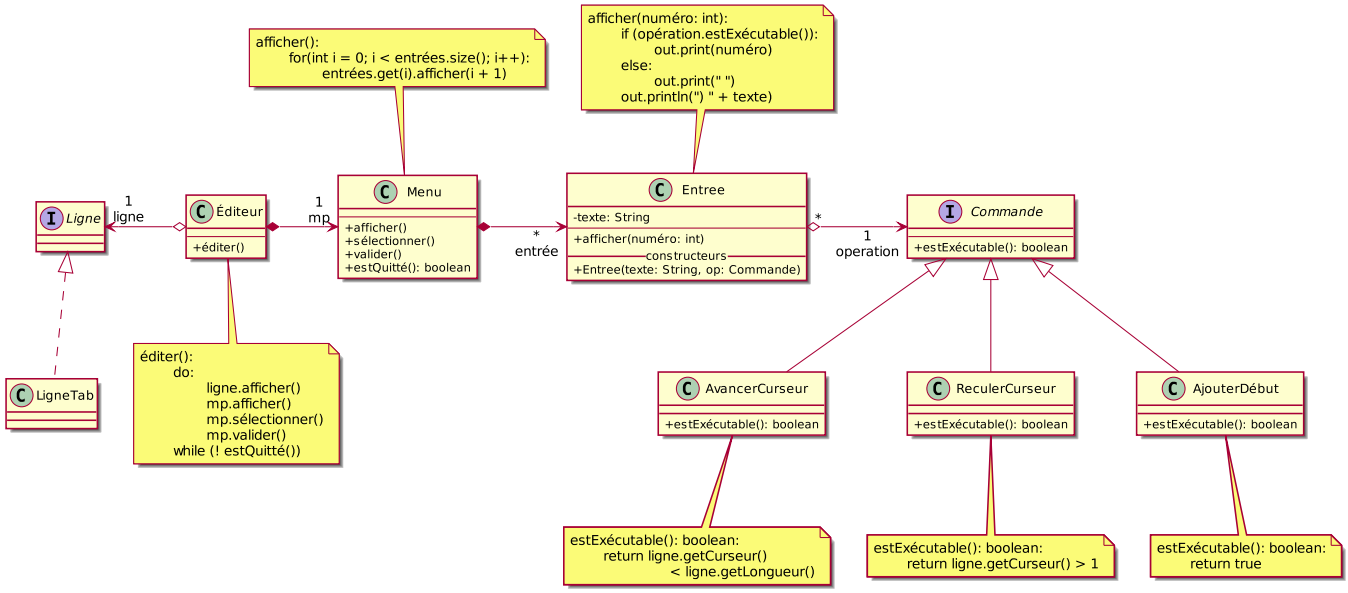
Comment faire pour définir les opérations de l'éditeur, par exemple « avancer le curseur », « ajouter un caractères au début » et « reculer le curseur » ?

Le principe est de réaliser l'interface `commande` pour chaque opération de notre éditeur d'une ligne et donc de définir sa méthode `estExécutable`.

Par exemple, on peut écrire une commande `AvancerCurseur` qui correspond à l'opération « avancer le curseur ».



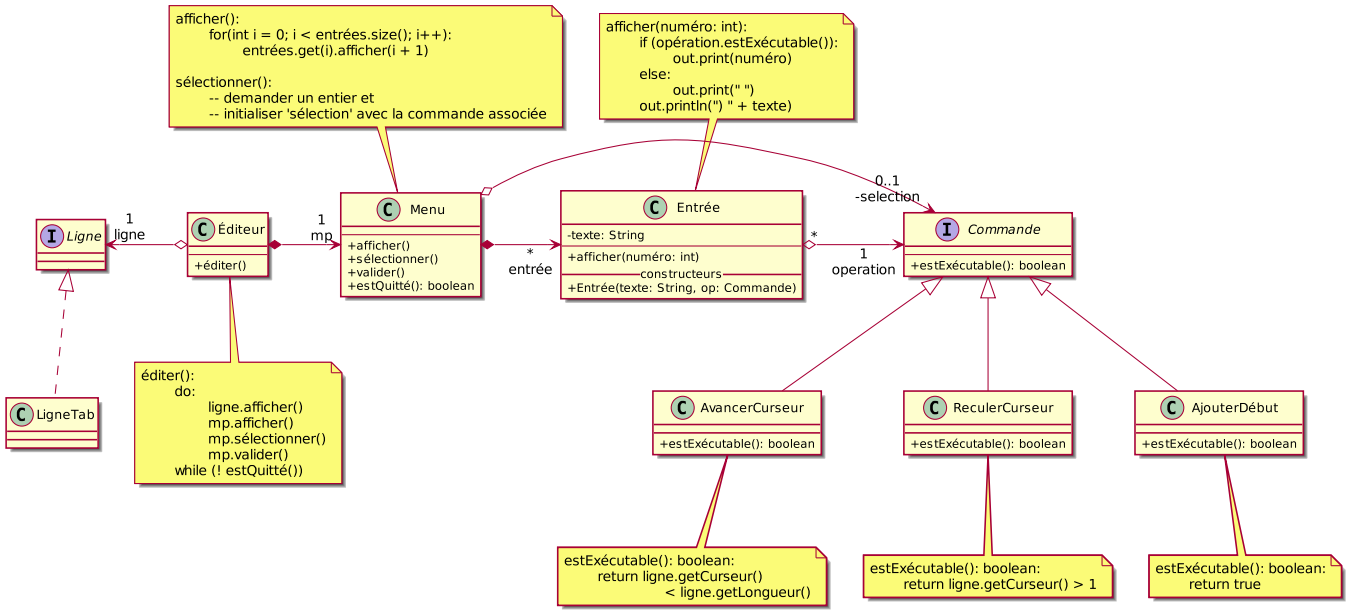
Et on pourra définir une classe `ReculerCurseur`, nouvelle réalisation de l'interface `Commande`, pour l'opération « reculer le curseur ». Et pour « ajouter au début de la ligne », on pourra définir `AjouterDébut`, autre réalisation de `Commande`. Pour toutes ces classes, on définit la méthode `estExécutable`.



Nous n'avons pas représenté la relation qu'il y a entre les réalisations de `Commande` que nous venons d'écrire et l'interface `Ligne`. Nous y reviendrons plus tard...

On vient de finir d'écrire `Menu.afficher` et tout ce qui lui était nécessaire.

Passons à `Menu.sélection`. Comme dans la première version, il s'agit de demander un entier à l'utilisateur. Ici l'entier correspondra à une commande. On peut ajouter une autre relation entre `Menu` et `Commande` avec comme rôle « sélection », multiplicité de 0..1 et droit d'accès privé.

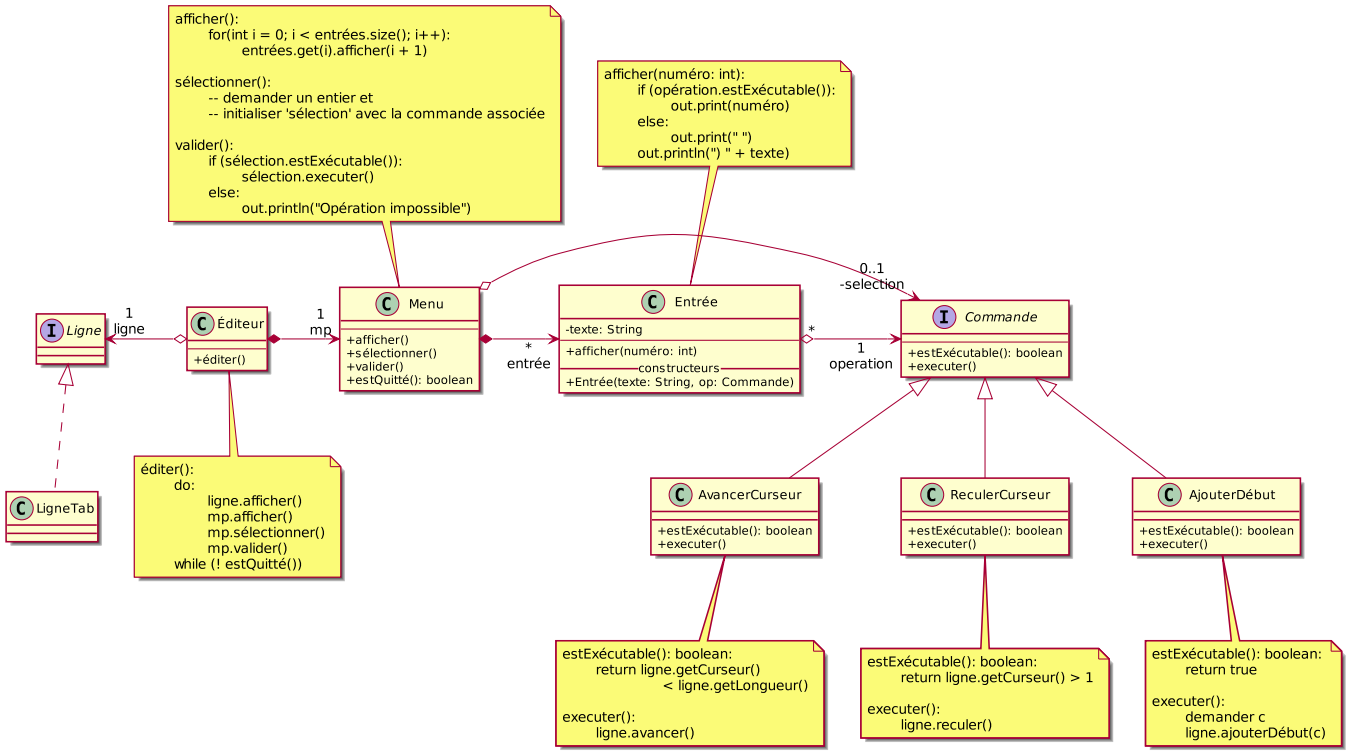


Il nous faut aussi écrire la méthode `Menu.valider`. Comment faire ?

Valider consiste à exécuter la commande sélectionnée si elle est exécutable. Puisqu'on doit « exécuter la commande », on va spécifier une méthode `exécuter` sur `Commande`. Elle ne pourra être appelée sur une commande que si cette commande est exécutable. Comme pour `estExécutable`, on ne sait pas écrire son code tant qu'on ne connait pas l'opération modélisée par cette commande. C'est donc une méthode abstraite et `Commande` peut rester une interface.

Bien sûr, il faut définir la méthode `exécuter` dans toutes les réalisations de `Commande` : `AvancerCurseur`, `ReculerCurseur`, `AjouterDébut`, etc.

Voici le diagramme de classe complété.



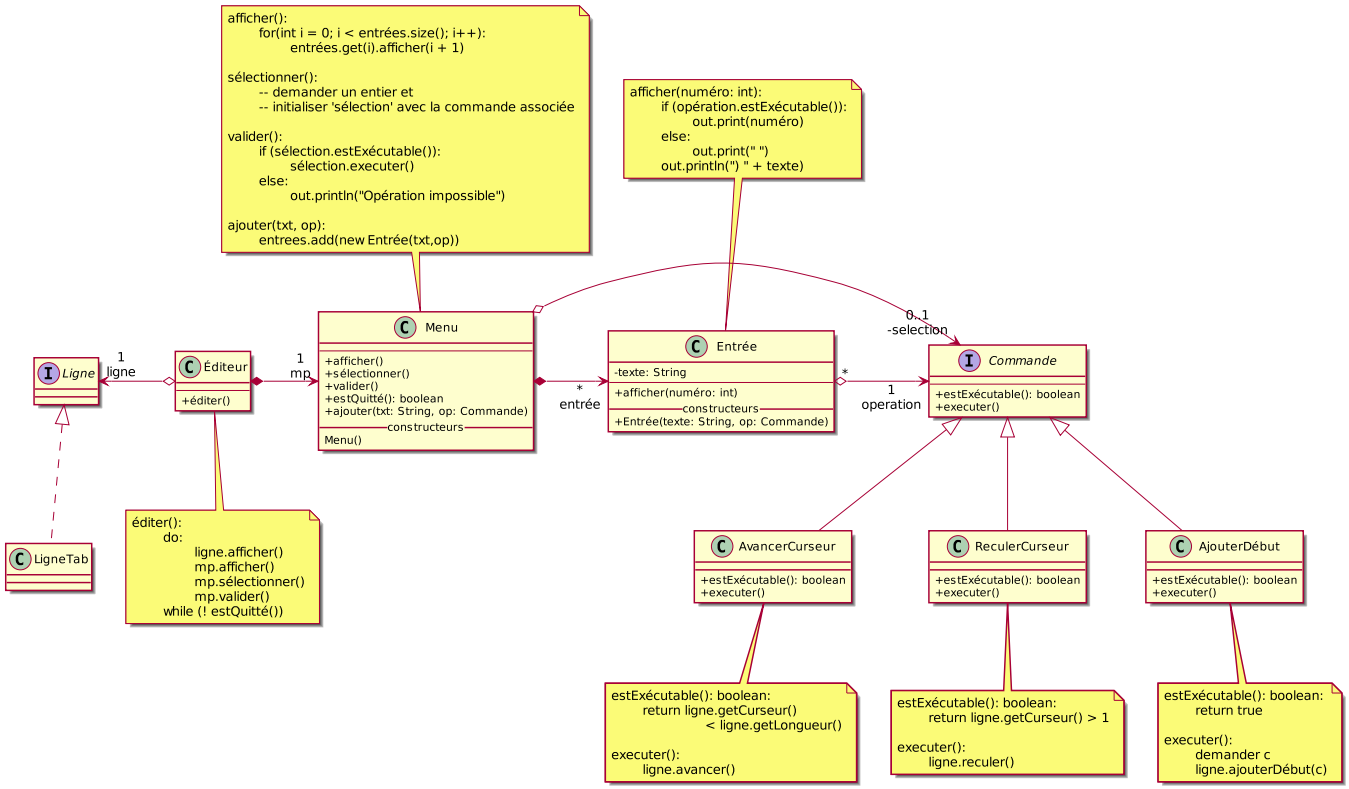
Nous ne détaillons pas la méthode `Menu.estQuitté` : indique si l'utilisateur a choisi 0.

Nous avons défini les méthodes que nous avions identifiées sur `Menu`. Mais cette classe n'est pas pour autant finie...

1. Quel constructeur sur `Menu` ?
2. Est-ce qu'il manque des méthodes sur `Menu` ?

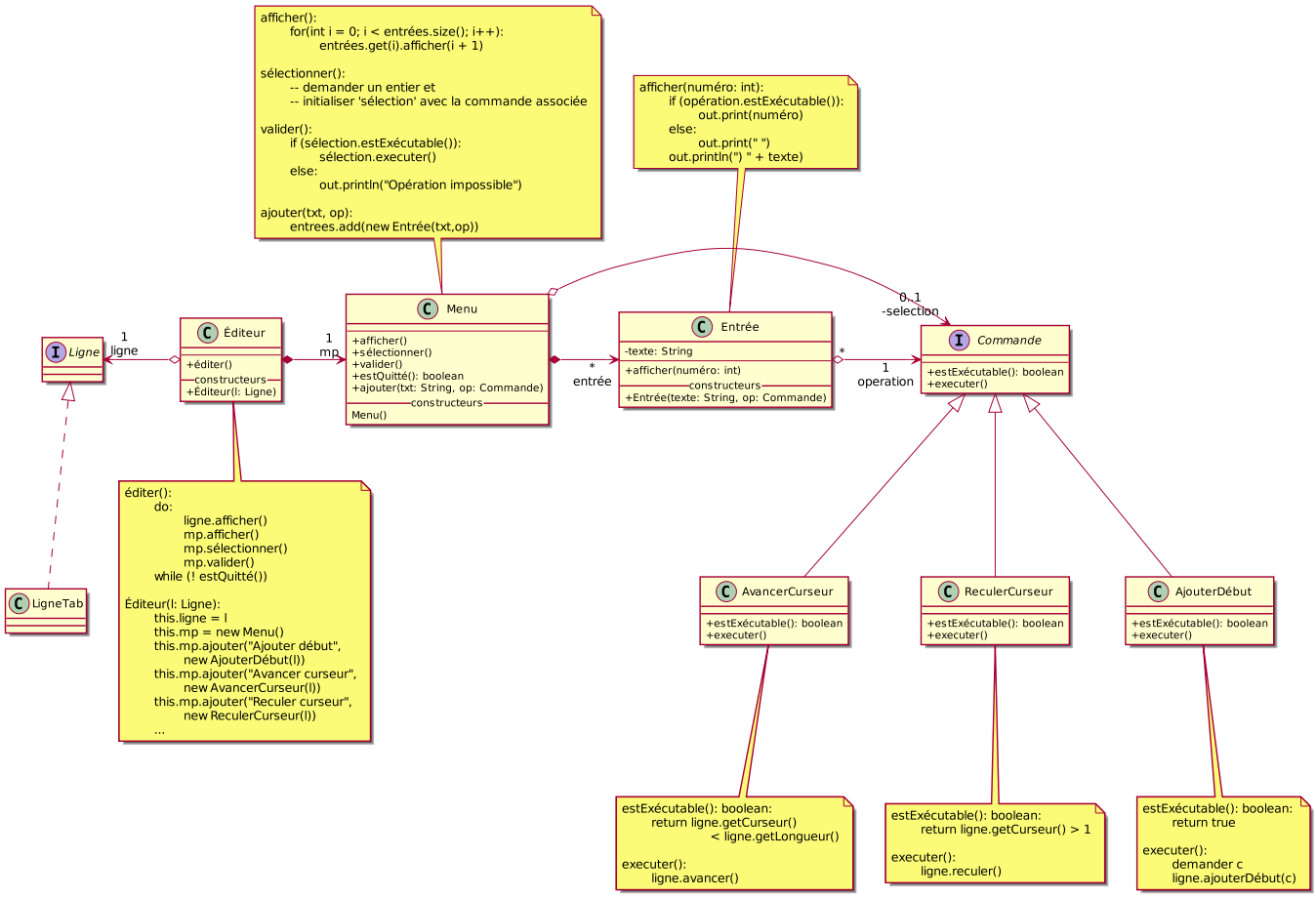
Le constructeur de `Menu` va créer un menu vide. On pourrait imaginer qu'il prenne en paramètre un titre.

Sur `Menu`, il manque donc une opération : celle qui permettra de lui ajouter des entrées puisqu'au départ le menu sera vide. Est-ce que l'on veut que l'utilisateur manipule des entrées ? Non. `Entrée` est plutôt une classe locale qui pour seul objectif de regrouper texte et commande et proposer les opérations pour les manipuler (`afficher`). Aussi, on définit sur `Menu` un méthode `ajouter` qui prend en paramètre le texte qui sera affiché dans le menu (`texte`) et la commande qui correspond à l'opération qui sera déclenchée (`op`). Son code consistera à créer une instance de `Entrée` et à l'ajouter à la fin de la liste des entrées du Menu.



Maintenant comment faut-il compléter le constructeur de l'éditeur pour construire le menu ?

Dans le constructeur de `Éditeur`, il faudra construire le menu principal pour y ajouter toutes les opérations de l'éditeur en précisant le texte associé et l'opération correspondante (appel à la méthode `Menu.ajouter`).



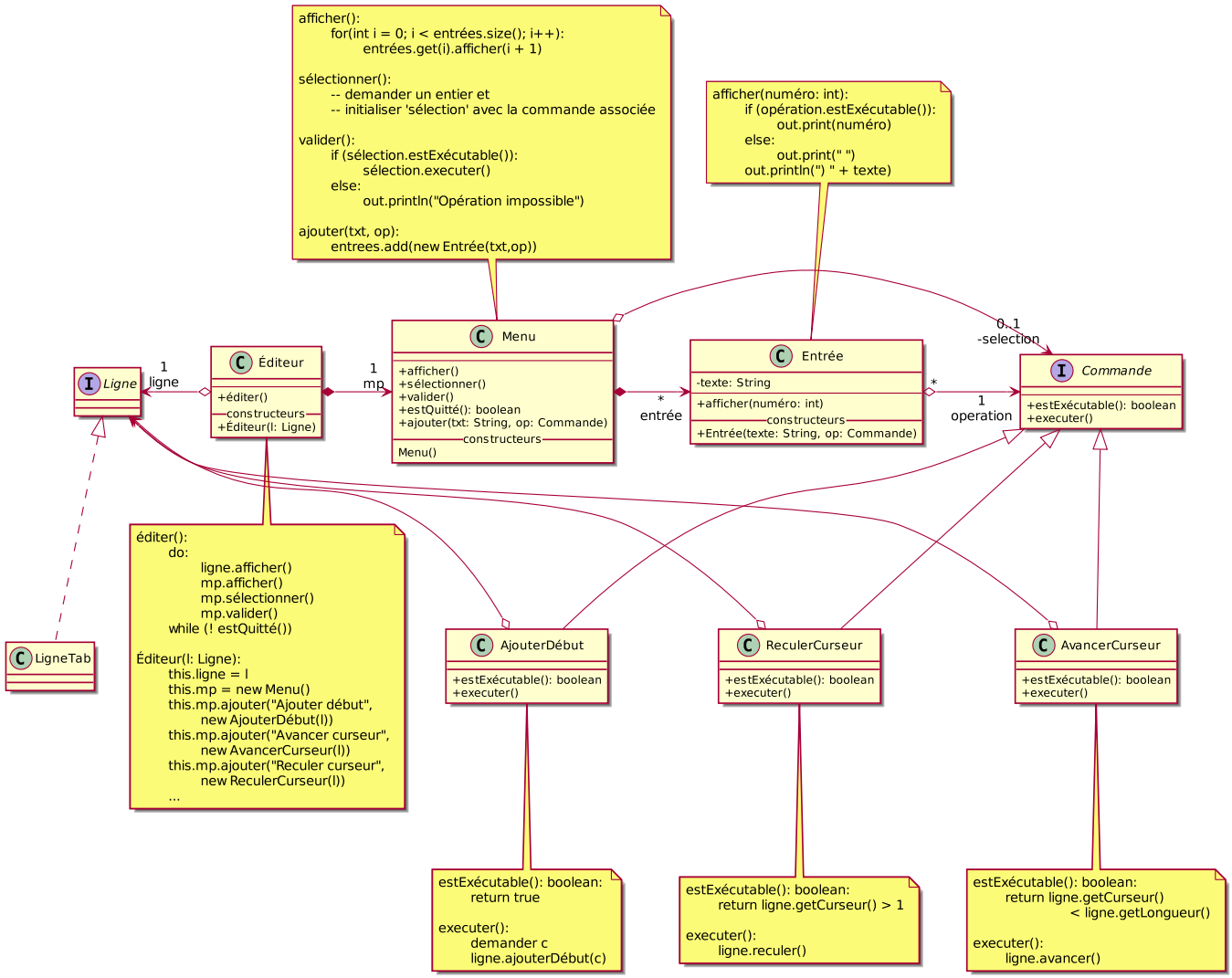
Revenons sur l'exercice 1.
La phrase qui résume ce qui est demandé arrive juste après son énoncé :

> « ...un éditeur orienté ligne en définissant des menus textuels réutilisables qui prennent en compte la notion d'opérations non réalisables. »

Elle résume bien ce qui est demandé, sauf l'aspect « réutilisable » qui n'est pas mentionné avant l'exercice 1.
Elle est très proche de la solution mise en œuvre (dans la deuxième version de l'éditeur d'une ligne et de ses menus).

Revenons sur la relation entre une commande de notre éditeur et l'interface `Ligne`.
On remarque que toutes les commandes qui correspondent à une opération sur la ligne vont avoir une relation d'agrégation (ou d'association) sur `Ligne`. Cette relation n'était pas représentée sur les diagrammes de classe précédents car elle aurait rendu les diagrammes moins lisibles, surtout si les commandes se multiplient (10 dans le sujet, beaucoup plus si on veut représenter

toutes les commandes d'un éditeur de texte). Le diagramme suivant en est la preuve (même s'il ne présente que trois commandes !).



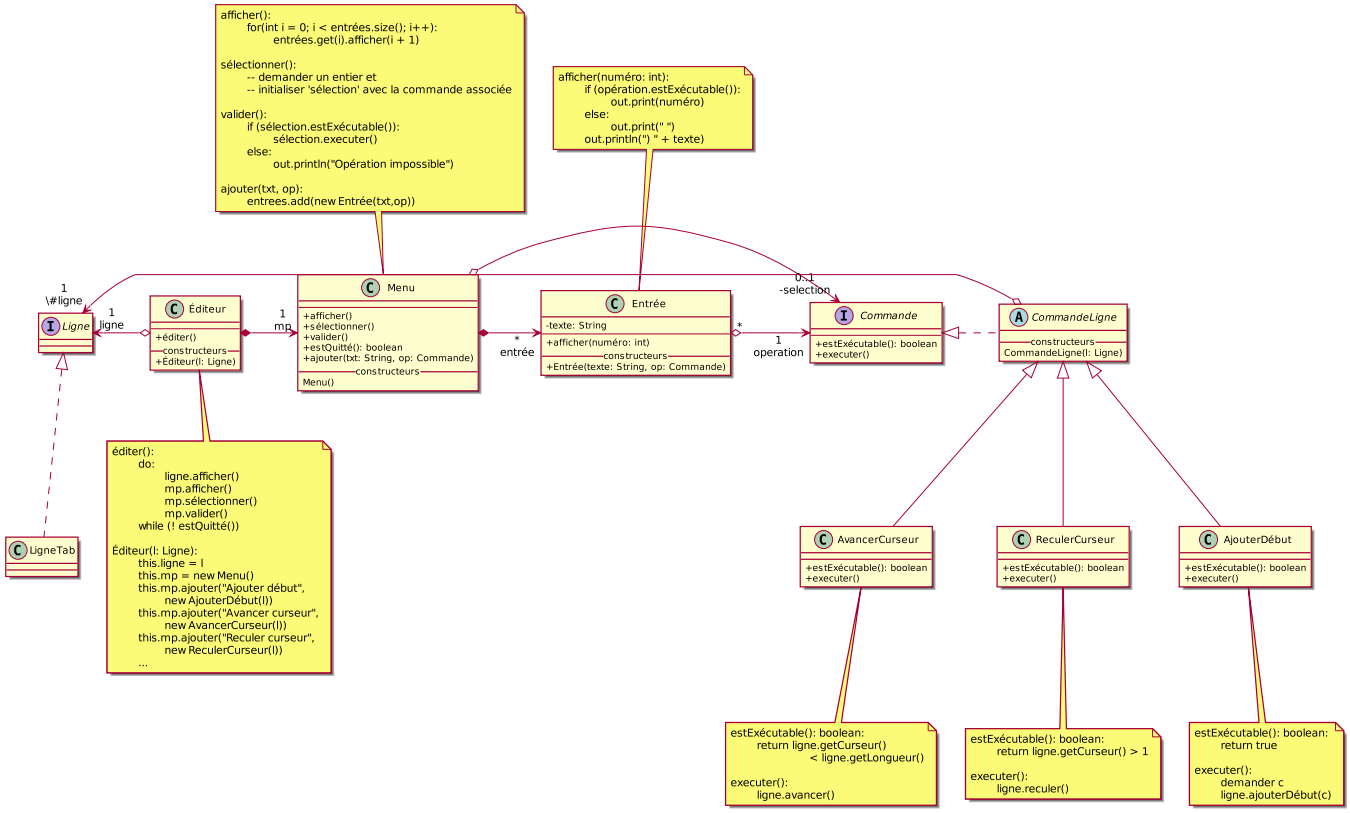
Comment éviter ce problème ?

On peut donc factoriser cette relation d'agrégation dans une classe `CommandeLigne` avec un attribut `Ligne` et un constructeur pour l'initialiser. Cette classe est abstraite puisque la méthode `estExécutable` n'est toujours pas définie. Sa relation vers l'interface `Ligne` se traduit en Java par un attribut déclaré **`protected`** car il est destiné à être utilisé par ses sous-classes.

Outre la diminution des relations sur le diagramme UML, cette classe est surtout intéressante pour classifier les commandes. C'est le principe que l'on retrouve sur les exceptions avec `Error` et `Exception` qui permettent une classification des `Throwable`.

Si on veut utiliser le menu textuel pour une nouvelle application, on aura certainement une classe abstraite `CommandeNouvelleAppli`.

Même pour PlantUML, l'outil utilisé pour produire ces diagrammes, on a résultat plus satisfaisant grâce à l'introduction de la classe abstraite `CommandeLigne`.



Exercice 6 : Évaluation de l'éditeur et de ses menus

Pour évaluer l'application écrite dans l'exercice 5, nous allons envisager plusieurs extensions et/ou modifications qui pourraient être demandées par notre client (celui qui nous a demandé de développer cette application).

6.1. *Ajouter de nouvelles opérations.* Comment faire pour ajouter une nouvelle opération sur l'éditeur (et donc une nouvelle entrée dans le menu) ?

Solution : Pour ajouter une nouvelle opération, il suffit de :

1. Écrire la nouvelle réalisation de `Commande` qui correspond à cette nouvelle opération en définissant les méthodes `estExecutable` et `executer`.

On écrit une nouvelle classe à côté des autres. Ainsi, on ne risque pas de casser quoique ce soit dans l'interface.

2. Ajouter une instance de cette nouvelle commande dans le menu principal de l'éditeur (dans son constructeur) avec le texte approprié.

Les modifications sont localisées et il y a peu de risque d'erreur en les réalisant.

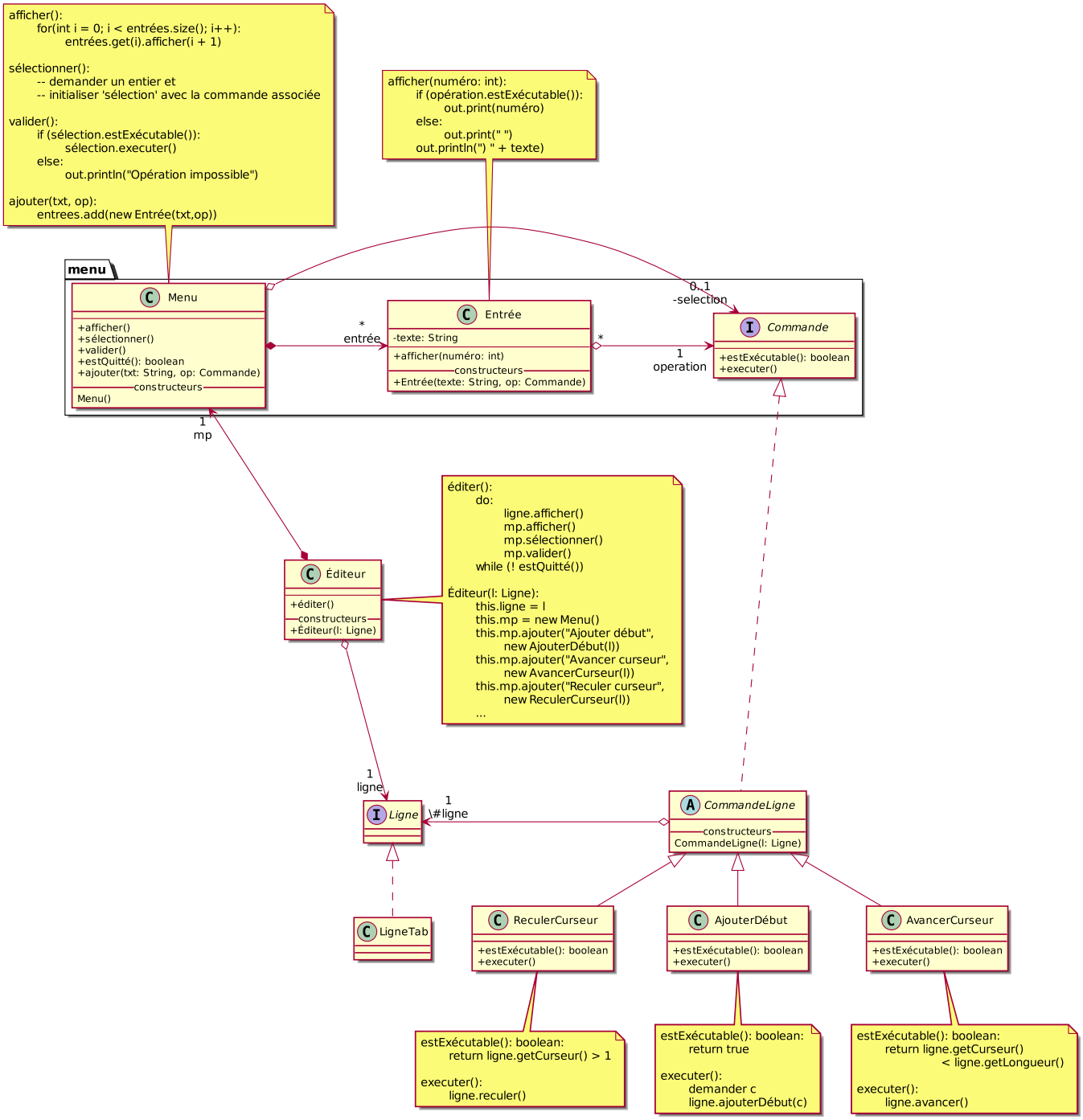
6.2. *Réorganisation des entrées du menu.* Comment modifier l'ordre des entrées dans le menu ?

Solution : Il suffit de changer l'ordre des `ajouter` dans la construction du menu dans le constructeur de `Éditeur`. Pas de risque d'introduire d'erreurs !

6.3. *Réutilisation du menu.* Que peut-on réutiliser concernant les menus si on doit développer une autre application qui utilise également des menus ?

Solution : On peut mettre `Menu`, `Entrée` et `Commande` dans un paquetage `menu`. Elles forment un tout qui compile et pourra être réutilisé dans différentes applications. Il suffira de définir les commandes de cette application et construire ses menus. Voir le diagramme de classe ci-après.

Notons que les flèches en UML indiquent bien le sens de dépendance. Les dépendances entre `Commande` et une commande particulière sont bien dans le sens « une commande particulière dépend de `Commande` ».



6.4. *Aide sur les opérations.* Lorsque l'utilisateur tape un numéro d'accès correspondant à une opération non réalisable, nous lui indiquons qu'il ne peut pas choisir cette opération. Il serait plus agréable pour l'utilisateur de savoir pourquoi l'opération n'est pas réalisable. Comment modifier la gestion des menus pour que cette explication puisse être donnée à l'utilisateur ?

De la même manière, quand l'utilisateur sélectionne une entrée du menu, il serait possible de lui afficher une « bulle d'aide » qui lui explique brièvement ce que fait l'opération associée.

Solution : Il suffit d'ajouter une méthode abstraite `aide` dans `Commande` qui retourne une chaîne de caractère et de la définir dans toutes ses réalisations. On ne peut pas oublier de la définir dans une classe car le compilateur nous signalera une erreur en disant que la classe devrait être déclarée abstraite car la méthode `aide` est abstraite.

6.5. *Raccourcis clavier.* Comment faire pour définir un raccourci clavier pour accéder aux opérations (en plus des numéros) ? Ce sont les lettres entre crochets sur les figures 1 et 2.

Solution : Il faudra l'écrire dans `Menu`. Par exemple, on peut surcharger la méthode `ajouter` avec un troisième paramètre qui est le raccourci. On pourrait vérifier qu'il n'est pas déjà utilisé et le conserver pour qu'il soit exploité par `sélection`.

Il faut travailler mais ce travail ne sera fait qu'une fois dans le paquetage `menu` et réutilisé par toutes les applications qui utiliseront ces menus textuels.

6.6. *Structuration du menu en sous-menus.* On constate que le menu est trop grand et on souhaite regrouper les opérations par thème avec, par exemple, un menu spécifique pour les opérations liées au curseur.

Le menu proposé ne comporte qu'une petite partie des opérations de l'éditeur et il est déjà long. Il serait donc souhaitable de pouvoir regrouper les opérations par thème et de les structurer en sous-menus, par exemple en regroupant dans un sous-menu les opérations relatives au curseur. L'application propose alors un menu principal et des sous-menus.

Comment faire pour intégrer cette notion de menus et sous-menus dans notre éditeur ?

Remarque : Il existe deux types de sous-menu : les sous-menus qui ne permettent la sélection que d'une seule opération et qui disparaissent et les sous-menus qui restent affichés jusqu'à ce qu'ils soient quittés explicitement par l'utilisateur. Ces derniers sous-menus permettent de sélectionner plusieurs opérations.

Solution : Ce sera l'objet du premier TP associé à ce thème.

6.7. *Pouvoir annuler une opération.* En plus de griser les opérations non réalisables, un autre aspect important pour une interface homme/système est la possibilité d'annuler la ou les dernières opérations réalisées. Ceci permet à l'utilisateur de faire des essais sans risque. L'éditeur devra permettre de défaire (et refaire) les dernières opérations exécutées.

Expliquer comment il serait possible de modifier l'éditeur pour autoriser l'annulation des opérations réalisées.

Solution : Ce sera l'objet du deuxième TP sur ce thème.

6.8. *Modification dynamique des entrées du menu.* Comment faire pour ajouter ou supprimer des entrées du menu pendant l'exécution d'un programme ?

Remarque : Ceci peut être contradictoire avec les aspects ergonomie. En effet, si la structure du menu change, l'utilisateur peut ne plus s'y retrouver. Dans ce cas, il est préférable de griser l'entrée plutôt que de la supprimer. Construire dynamiquement un menu est en revanche utile pour gérer la liste des derniers fichiers édités, par exemple.

Solution : On peut appeler la méthode `ajouter` de `Menu` pour ajouter dynamiquement des entrées au menu. Par exemple, l'équivalent des fichiers récemment ouverts. On pourrait aussi prévoir une opération pour supprimer une entrée.

Conclusion : On vient de voir deux manières de traiter les menus textuels. Les comparer.

La première est simple à comprendre mais difficile à faire évoluer et non réutilisable.

La deuxième est plus compliquée mais elle est réutilisable et si on doit modifier les menus de nos applications, il y a peu de risque de faire d'erreurs. Même si cette deuxième solution est plus compliquée, il me semble qu'on y arrive assez naturellement.

D'ailleurs, si vous aviez voulu reprendre la première version pour éviter les « ... » que nous avions utilisés, vous seriez arrivés à une solution équivalente. Par exemple pour afficher le menu avec une boucle (pour éviter les « ... »), on aurait pu mettre les textes dans un tableau ou une liste. Le numéro aurait été la variable de boucle. Pour savoir s'il fallait afficher le numéro ou un tiret, on aurait pu mettre un booléen dans un tableau mais alors il aurait fallu le mettre à jour à chaque modification de la liste (pas si simple). Une alternative est de déclarer un tableau des fonctions, chaque fonction nous disant s'il faut mettre ou non le numéro. La fonction est l'équivalent de notre méthode `estExecutable`. Enfin, dans traiter il aurait fallu exécuter le code correspondant au numéro choisi. Pour éviter le **`switch`** on aurait donc certainement défini un tableau de procédures qui correspondent à l'action à exécuter (l'équivalent de notre méthode `exécuter`).

On se retrouve alors avec plusieurs tableaux, l'un de textes, l'autre de fonctions équivalentes à `estExécutable` et le dernier de procédures équivalentes à `exécuter`. Pour éviter les incohérences entre les tableaux, on pourrait faire un seul tableau dont le type serait un enregistrement regroupant les trois informations précédentes. Ce type enregistrement est en fait notre interface `Commande` (sauf que nous n'avions pas mis le texte dans `Commande`).

On pourrait donc facilement implanter cette solution dans un langage impératif tel que Ada, C, Python, etc. sans utiliser l'approche objet. On peut s'en sortir avec des « pointeurs sur des sous-programmes ».

Quels sont les apports de l'approche objet alors ?

1. La structuration : raisonner objet oblige à bien identifier les « modules » (ici les classes) de l'application et à clairement définir leurs responsabilités.
2. La localité des données. Considèrons le fait de pouvoir annuler les opérations de l'éditeur. En le traitant un peu vite, annuler un commande, c'est donc avoir une méthode `annuler` sur `Commande`. Pour annuler, on pourrait modifier à chaque fois toute la ligne pour la restaurer quand on annule (c'est lourd, surtout si on édite de gros fichiers). Une alternative consiste à mémoriser les informations qui seront nécessaires à annuler les effets de la commande.

   Par exemple, pour annuler l'opération « supprimer le caractère sous le curseur », il faut mémoriser le caractère supprimé pour le remettre à sa place quand on annulera. La question est de savoir où conserver cette information ? D'autant plus que des opérations différentes auront besoin de conserver des informations différentes.

   En objet, on le stockera tout naturellement dans la classe de la commande correspondant à l'opération (sous la forme d'un attribut). Chaque réalisation de `Commande` pourra stocker les informations qu'elle souhaite.

   Ce serait plus difficile à faire en C, Ada ou Python (sans utiliser les aspects objets de ces derniers langages).

Dernier mot : Au-delà du résultat (de l'architecture à laquelle nous sommes arrivés), retenez la démarche que nous avons mise en œuvre !