Un éditeur orienté ligne

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.

Exercice 1 : Comprendre le problème posé
Résumer en une phrase l'application à développer.

```
~
------------------------------------------------------
 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)

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

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

Listing 1 – La spécification d'une ligne

```
/** Spécification d'une ligne de texte.
  * @author  Xavier Crégut (cregut@enseeiht.fr)
  */
public interface Ligne {
      //@ public invariant 0 <= getLongueur(); // La longueur est positive
      //@
      //@ // Le curseur est toujours sur un caractère sauf si la ligne est vide.
      //@ public invariant 0 <= getCurseur() && getCurseur() <= getLongueur();
      //@ public invariant getCurseur() == 0 <==> getLongueur() == 0;

      /** nombre de caractères dans la ligne */
      /*@ pure @*/ int getLongueur();

      /** Position du curseur sur la ligne */
      /*@ pure @*/ int getCurseur();

      /** le ième caractère de la ligne
       * @param i l'indice du caractère
       * @return le ième caractère de la ligne
       */
      //@ requires 1 <= i && i <= getLongueur(); // indice valide
      /*@ pure @*/ char ieme(int i);

      /** Le caractère sous le curseur
       */
      //@ requires getLongueur() > 0; // la ligne est non vide
      /*@ pure @*/ char getCourant();

      /** Avancer le curseur d'une position à droite. */
      //@ requires getCurseur() < getLongueur();  // pas à la fin
      //@ ensures getCurseur() == \old(getCurseur()) + 1; // curseur avancé
      void avancer();

      /** Avancer le curseur d'une position à gauche. */
      //@ requires getCurseur() > 1;      // pas en début de ligne
      //@ ensures getCurseur() == \old(getCurseur()) - 1; // curseur reculé
      void reculer();

      /** Placer le curseur sur le premier caractère. */
      //@ requires getLongueur() > 0; // ligne non vide
      //@ ensures getCurseur() == 1; // curseur sur la première position
      void raz();

      /** Remplacer le caractère sous le curseur par le caractère c. */
      //@ requires getLongueur() > 0;
      //@ ensures getCourant() == c;
      void remplacer(char c);

      /** Supprimer le caractère sous le curseur. La position du curseur reste
       * inchangée.
       */
      //@ requires getLongueur() > 0;
      //@ ensures getLongueur() == \old(getLongueur()) - 1; // un caractère ôté
      //@ ensures getCurseur() == Math.min(\old(getCurseur()), getLongueur());
      void supprimer();

      /** Ajouter le caractère c avant le curseur.
       * Le curseur reste sur le même caractère.
       */
      //@ requires getLongueur() > 0;  // curseur positionné
      //@
```

```
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) ?

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

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

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.

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.

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.

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.

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.