Introduction

The text editor program is designed to highlight certain aspects of traditional, and object-oriented, programming. We will argue that these are aspects that we would like to avoid and the demonstration will motivate a justification for moving towards functional programming. You may or you may not be wholly convinced by the argument but be aware of being biased by your experience. Perhaps you might reserve judgement until you have completed this module.

The model is simple – too simple for a real text editor. However, it contains the basic components of such an application and it is easy to see how the model could be built upon.

A basic editor maintains a sequence of character elements. There is the notion of a cursor which lies either at the start of the text, the end of the text, or between two characters within the text. This can be displayed viz:

[abc|def] – The string “abcdef” with the cursor (position 3) between the ‘c’ and the ‘d’.

[abcdef|] – The string “abcdef” with the cursor (position 6) at the end of the string.

[|abcdef] – The string “abcdef” with the cursor (position 0) at the start of the string.

[|] – an empty string with the cursor at position 0.

The use of ASCII to represent the state means that it is easy to view on a console (standard output) and easy to check the state. Since the cursor is represented by a ‘|’ character, we will avoid entering this character into the string: the model is only an example.

The text editor will focus on standard operations such as insertion and deletion and moving the cursor position. Specifically:

- **backspace**() // deletes the character to the left of the cursor (if there is one)
- **delete**() // Deletes the character to the right of the cursor (if there is one)
- **insert**(c: Char) // Inserts a new character c at the cursor position and advances the cursor so that it lies just after the newly-inserted character
- **left**() // Moves the cursor one place to the left (or does nothing if at start of line)
- **modify**(c: Char) // Changes the character to the right of the cursor to c
- **right**() // Moves the cursor one place to the right (or does nothing if at end of line)

Later the model will be extended to provide an (infinite) undo operation.

Basic editor design

The basic editor maintains a sequence of character elements. This is represented by the aggregation in the following class diagram.

The **CharElement** class maintains individual characters as objects which may have state. This permits a variety of attributes to be stored about each character including, for example, font-colour, font-type, font-size, bold, italic, underlined, etc.

```
class CharElement(initChar: Char) {

  private var c: Char = initChar

  def setChar(c: Char): Unit =
    this.c = c

  def getChar: Char = c

  override def toString: String = s"$c"
}
```

The class **CharElement** encapsulate the character's (private) state and exports a setter method and a getter method. It also overrides **toString**. *(NB: The class could contain other private data fields to represent font-colour, etc., as described above, along with getters and setters for each of these. We have only stored the character value in this example in order to keep the model simple.)*

The **BasicEditor** class maintains the collection of character elements as a sequence using Scala's built-in, mutable **ListBuffer** class.

```
class BasicEditor {

  import collection.mutable.ListBuffer

  /** The text is stored as a mutable sequence of character elements */
  protected var text: ListBuffer[CharElement] = new ListBuffer[CharElement]

  /** The number of characters currently on the line */
  protected var size: Int = 0

  /** The cursor position which can range between 0 and size */
  protected var cursor: Int = 0
```

Note that the fields are marked as protected rather than private. This means that its subclasses will have access to these data fields. However, outside of this class and its subclasses, the internal state representation is not visible and data hiding (encapsulation) is established.

The **BasicEditor** has three methods for manipulating the data structure. These methods are also marked as protected so they are visible to this class and its subclasses but not outside. These methods simply delegate to the **ListBuffer** class by calling the relevant methods on **text**.

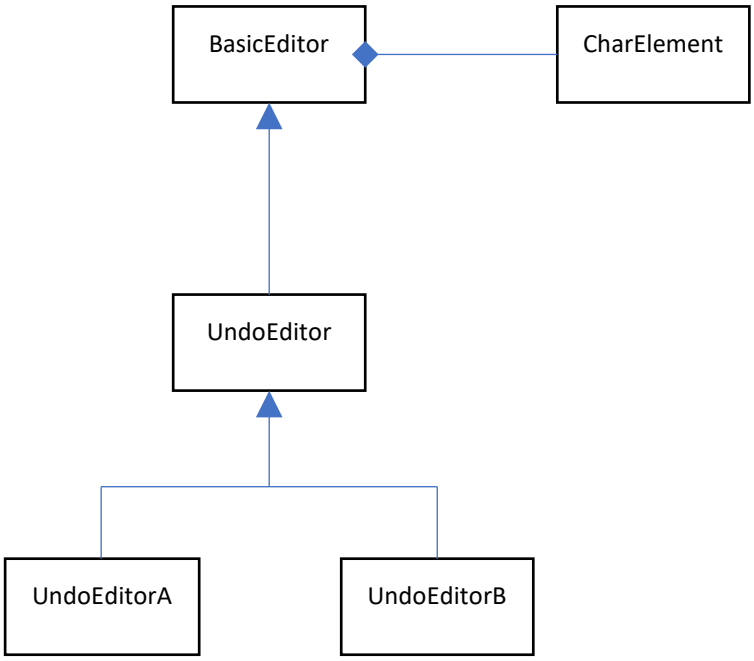
```
protected def insertCharacter(position: Int, ch: CharElement): Unit = {
  text.insert(position, ch)
}

protected def deleteCharacter(position: Int): Unit = {
  text.remove(position)
}

protected def modifyCharacter(position: Int, f: CharElement => Unit): Unit = {
  f(text(position))
}
```

The public interface is provided to supply each of the editor operations described in the introduction. Within these public methods all manipulation of the data structure is achieved by calling the (protected) methods shown above.

```
final def left(): Unit = {
  if (cursor > 0) cursor = cursor - 1
}

final def right(): Unit = {
  if (cursor < size) cursor = cursor + 1
}

final def insert(c: Char): Unit = {
  insertCharacter(cursor, new CharElement(c))
  cursor = cursor + 1
  size = size + 1
}

final def delete(): Unit = {
  if (cursor < size) {
    deleteCharacter(cursor)
    size = size - 1
  }
}

final def backspace(): Unit = {
  if (cursor > 0) {
    left()
    delete()
  }
}

final def modify(c: Char): Unit = {
  if (cursor < size)
    modifyCharacter(cursor, _.setChar(c))
```

Undo editor

To perform successive undo operations requires the history of the editor state to be maintained in a stack. Each undo pops the stack once and replaces the current state with the one just popped. In all other respects an **UndoEditor** is a **BasicEditor** and so the former is designed to inherit this functionality from the latter. The class diagram below shows the inheritance relationship between **BasicEditor** and **UndoEditor**.



Note that the class diagram also has two further subclasses of the **UndoEditor**. Each of these overrides one specific piece of behaviour (i.e. a method) in the **UndoEditor** and consequently each of **UndoEditorA** and **UndoEditorB** provides a specific variation in the undo behaviour. This will be discussed at the end.

The **UndoEditor** only needs to maintain the stack of previous editor states. Therefore, a local variable called **history** is created as an instance of a **ListBuffer** and accessed using the stack protocol. The **history** records snapshots of the **text** buffer.

```
class UndoEditor extends BasicEditor {

  import collection.mutable.ListBuffer

  protected val history =
    new collection.mutable.ListBuffer[ListBuffer[CharElement]]

  protected def pushCurrentState(): Unit = {
    history.insert(0, text)
  }
```

```
  def undo(): Unit = {
    if (history.nonEmpty) {
      size = history.head.size
      cursor = scala.math.min(size, cursor)
      text = history.head
      history.remove(0)
    }
  }
```

The **undo**() method is noteworthy. Provided the stack is non-empty, then the top of the stack is accessed directly and the values of **text**, **size**, and **cursor** are updated accordingly. Note the use of the **scala.math.min** function – if the popped string is shorter than the current one then the cursor may point beyond the restored state. In this case the cursor is repositioned to the end of the restored string. Finally, the top of the stack is removed, effecting a stack *pop* operation.

The protected data-structure-accessing methods from **BasicEditor** can now be overridden in **UndoEditor** so that each will *push* (save) the current state onto the stack before delegating the actual update operation (up) to the superclass.

```
override protected def insertCharacter(position: Int, ch: CharElement): Unit = {
  pushCurrentState()
  super.insertCharacter(position, ch)
}

override protected def deleteCharacter(position: Int): Unit = {
  pushCurrentState()
  super.deleteCharacter(position)
}

override protected def modifyCharacter(position: Int, f: CharElement => Unit): Unit
= {
  pushCurrentState()
  super.modifyCharacter(position, f)
}
```

Note the type of parameter **f** in **modifyCharacter**(). This defines a function being passed as an argument to the method. This is a brief example of the functional style to which we turn later.

UndoEditorA

UndoEditorA is exactly the same as **UndoEditor** except for one piece of functionality: the **pushCurrentState**() method is overridden:

```
override protected def pushCurrentState(): Unit = {
  history.insert(0, text.clone())
}
```

The effect of using **text.clone**() rather than **text** itself is discussed in the videos that accompany this case study.

UndoEditorB

UndoEditorB similarly extends **UndoEditor** with an overridden implementation of **pushCurrentState**():

```
override protected def pushCurrentState(): Unit = {
  history.insert(0, text.map(ch => new CharElement(ch.getChar)))
}
```

Once again, the effect of this modification is discussed in the videos that accompany this case study. See also how the parameter to the **map** method is a function (of the form **a => b**). We will see more of this functional style parameter passing throughout the module.