Ergonomics and HMI concerns in Mule: *towards intelligent input methods*

Didier Verna
The XEmacs Project
<didier@xemacs.org>

April 19, 1999

Abstract

This paper presents some ergonomics and Human Machine Interaction problems that several input methods (notably the French ones) introduce in Emacs [1]. First, a general overview of the available input methods is given. Then, some ergonomics problems are described. Finally, some ideas are proposed to improve the quality of these input methods.

Introduction

Working with an internationalized text editor means that not only several languages can be displayed, possibly at the same time, but also that those languages can be *typed in*. In order to ease the process of typing in different languages with a standard keyboard, Mule offers the concept of "input method" which allow you to enter characters from different alphabets. The process of inputing characters from a traditional keyboard can be particularly complex, notably for large character sets like kanjis. Without going that far, we can already identify ergonomics problems with simpler methods like the ones used for inputing French. In this paper, we would like to demonstrate those problems, and propose ideas that could be used in order to make those input methods more "intelligent", and thus simpler to use. Section 1 proposes an overview of the different available input methods. Section 2 demonstrates the ergonomics problems that French input methods suffer from. Section 3 attempts to propose some ideas for improving the quality and the ergonomy of these input methods.

[1] the term "Emacs" refers to any flavor of the software, notably GNU Emacs or XEmacs

1 Different Kinds of Input Methods

This section provides a short overview of the different kinds of input methods available in Mule for inputing different languages, and gives more details on the Quail French input methods.

1.1 Input methods classification

Class 1: key bindings The first class of input method works by rebinding the keyboard keys to characters from another language. This is possible for small alphabets that can be represented on a normal keyboard. Russian input methods belong to this category. Each character can also be printed on the corresponding key to help finding them on the keyboard. As far as French is concerned, note that an input method emulating an AZERTY keyboard on a QWERTY one would belong to this category.

Class 2: key sequences When there is not enough keys on the keyboard, or when you don't want to rebind them, some input methods use key sequences to input characters. Latin-1 (notably for French) and Latin-2 input methods belong to this category. The next subsection will give a more detailed overview of the French input methods.

Combination of class 1 and class 2 Some languages such as Thai use a combination of the first and second class to input characters. In a first step, the keys are bound to vowels, consonants etc. and a sequence of such characters actually produce a composite one.

Class 3: external help Finally, in more complex cases, some help from an external program might be required. Japanese input methods belong to this category: after using a combination of the preceding cases to input, say, hiragana phonetically, an external program like canna-server proposes possible kanji translations for this word.

1.2 Quail French input methods

Quail provides several French input methods belonging to class 2 as described previously. "french-postfix" and "french-prefix" are the most widely used. Those input methods typically let you enter a character with an accent or a cedilla in two keystrokes. In french-prefix, you type the symbol first, while in french-postfix, you type the letter first. Figure 1 shows some examples of key sequences from those input methods.

Obviously, it is also possible to cancel the key sequence, if you really need the two characters one after each other. Under a french-postfix input method, this is usually achieved by typing the symbol twice. Under a

french-prefix	french-postfix
‘ + a → à	a + ‘ → à
’ + e → é	e + ’ → é
, + c → ç	c + , → ç
< + < → «	< + < → «
/ + c → ©	c + / → ©
/ + o → °	o + / → °

Figure 1: Examples of Quail French key sequences

french-prefix input method, the space bar terminates the sequence after the symbol has been typed in. This mechanism is illustrated on figure 2.

french-prefix	french-postfix
‘ + <space> → ‘ + a → ‘ a	a + ‘ → à + ‘ → a ‘
’ + <space> → ’ + e → e ’	e + ’ → é + ’ → e ’
/ + <space> → / + c → c /	c + / → © + / → c /
/ + <space> → / + o → / o	o + / → ° + / → o /

Figure 2: Cancelling Quail French key sequences

As you can see, those key sequences are very easy to remember. The symbols used in french-postfix do look like the corresponding accents or cedilla which makes their use very intuitive. Doubling the symbol in order to cancel the sequence is also something rather natural.

2 Cognitive problems in Quail input methods

Despite their appearant simplicity, those input methods actually introduce some cognitive problems that from time to time can make them more difficult to use that it seems at a first glance. Figure 3 shows some common mistakes issued with those methods. The first one is in a french-prefix context, the others are in french-postfix. Each case shows the sentence that was obtained, and the one that was expected.

Obtained	Expected
Lémpire contre attaque	L’empire contre attaque
Sois franç ça ne marche pas…	Sois franc, ça ne marche pas…
Utilisez plutôt ‘setlocalé	Utilisez plutôt ‘setlocale’
/us®local/sr©xemacs	/usr/local/src/xemacs

Figure 3: Common mistakes under Quail French input methods

As you can see, those mistakes are always related to key sequence cancelation. What happens is that the user *forgets* to cancel the key sequence, by either typing <space> (in french-prefix) or by doubling the symbol (in french-postfix). We think that the reason for this breakage is that the concept of key sequence cancelation is counter intuitive. As we have already said, mapping symbols to accents is very easy to remember and doesn't raise any problem. However, cancelling a key sequence means that you actually input something *wrong* and you must be aware of it, because afterwards, you will have the opportunity to correct it. For instance, if you want to input a "c" followed by a comma in french-postfix, you must have in mind that you will first input a "c" cedilla and *then* correct it to what you want by doubling the comma.

According to the way Quail was designed, the idea of doubling the symbol to cancel a key sequence is probably the best choice that could be done. Which is arguable is not the cancelation method, but really the fact that cancelation is needed. This also stands for the french-prefix method. As a result, we should try to determine how an input method could avoid cancelation. This is the purpose of the next section.

3 Proposed solutions

In order to eliminate the need for a cancelation method, we must accept the fact that the user can type two keys in different circumstances with different ideas in mind. For instance, the key sequence <c comma> sometimes means "give me a c cedilla",and sometimes means "give me a c and a comma". We should consequently find ways to make the input methods understand the different cases without requiring anything special from the user.

3.1 Static solution: using the context

In the second example of figure 3, the situation happens to be rather simple to correct: in French, a word cannot be ended with a c cedilla. Consequently, if a space immediately follows the c cedilla, we know for sure that the user actually wanted a c and a comma. This example shows that we could benefit from the "context" of the key sequence, in other words, the characters already present around the insertion point. Quail blindly uses key sequences, without knowing anything about the current context.

Consequently, the first solution we can think of is defining an input method by *character-key sequences* rather than by key sequences only. Consider the sample specification given in figure 4. This specification means that typing a comma when there is a c in the buffer should produce a c cedilla. However, typing a space when there is a c cedilla in the buffer

{c} + , → ç
{ç} + <space> → c , <space>

Figure 4: character-key specification example

should turn it into a c followed by a comma and a space.

There is still something not completely satisfactory with this technique. Namely, the fact that the c cedilla will still be generated in wrong cases, even temporarily. Although the user does not have to correct it by hand, it can still be annoying to see it appear. This problem can partially be solved by using more than a single character as the context. For instance, in French, we do not have any word containing the sequence <i r a ç>. Consequently, if we add the rule "{irac} + , → i r a c ," to the input method specification, we will get immediately the proper characters in that case.

As we can see, by extending the concept of key sequence to the concept of character-key sequence, which should not be very hard to implement, the current input methods could be considerably improved with respect to the cancelation problems. However, several issues remain problematic:

- There exist cases that cannot be corrected with this method. The next subsection presents some of them.
- Specifying all the possible cases similar to the "irac" example would be enormous. We cannot afford it, especially because the user should still have the possibility to customize his key sequences.

3.2 Dynamic solution: relationship with spell checking

One of the problems that the preceding solution cannot solve is ambiguousness. This means that at the time a character-key sequence is encountered, it is not necessarily possible to decide which action should be taken. For instance, in French, the sequence "L ' e n i" is undecidable, because the quote can really be an apostrophe, but it could also be a french-prefix sequence for a word beginning with "L é n i" (there are some). As a consequence, it is only possible to decide what to do after some more characters are typed in. In our example, the next character can be sufficient. For instance, if it is an "f", the key sequence necessarily represents an "é", and if it is a "v", it is necessarily a real apostrophe. Otherwise, the corresponding word does not exist in French.

It is important to notice that when we speak of solving ambiguousness, what we have to do in terms of implementation is actually to look up into a dictionary and see if such or such word does exist. This process is actually highly related to the notion of spell checking... This idea of dynamic checking can also be applied to the cases (described in the previous subsection) where the decision could be made immediately. Take again the "i r a

c" example described precedently. Instead of specifying explicitly the key sequence as proposed in the first solution, we can also look up into a dictionary to see if such a word is possible. This way, we don't have to specify every possible sequence in the input method, the decision is made based on a generic mechanism.

4 How far should we go?

If we push even farther the relationship between input method and spell checking, we can reach reach the concept of "adaptive" input method and even the concept of word completion. However, going that far in the automation of character input is not necessarily a good thing.

4.1 Adaptive input methods

Consider the sequence (already typed) "C é r". In French, it is not possible that this sequence is followed with an "e". Only a "é" can happen. Knowing this, we can imagine that if the user actually types an "e", it could be automatically transformed into an "é". Here, we have just reached the concept of "adaptive" input method, that is a method mixing different classes (see section 1). Our method which is normally of class 2 turns out to be a class 1 method (key bindings) in that case, since the "e" key produces directly an "é".

4.2 Word completion

While we are at it, why stopping at the accents or cedilla level? Since the input method can sometimes decide what is the next character in a word, it can probably also know how to complete the whole word in some cases. There, we have gone from the concept of input method, through the notion of spell checking and finally got a word completion mechanism. This demonstrates how far those notions are related to each other.

So, how far should we go? Although only experiments with Mule users could give us a definitive answer, the ideas of adaptive input methods and word completion are probably not good things to implement. In Human Machine Interaction, we know that automation is good only if the user perceives it as a stable behavior. It is highly probable that if an input method sometimes transforms an "e" directly into an "é" and sometimes does not, the user will be annoyed rather than pleased, because it is out of his capacity to remember exactly in which cases the first behavior happens and in which cases the second one takes place. The same consideration applies for the notion of word completion.

conclusion

In this paper, we have described briefly the main classes of input methods, and given a more detailed view on the Quail French ones which belong to the second class. The concept of key sequence cancelation, although necessary given the way Quail is designed, appears to be counter intuitive and is at the origin of numerous breakages in the process of inputing characters. While examining possible solutions to eliminate the need for a key sequence cancelation process, we have finally demonstrated that the notion of input method, at least in the case of French, and probably for all Latin languages, is deeply related to spell checking. If one day we can make spell checkers understand that "e ' " is actually a *misspelled* version of "é", then flyspell will probably be the most efficient input method ever written.

However, we should keep in mind that even spell checking input methods will not solve all the problems. In the example of "/us®local/sr©xemacs", a broken pathname, the only way to correct it automatically would be for the machine to *know* that this is a pathname, and that the way it is currently written is meaningless. However, nowadays, meaning recognition is still another story...