Commands Composition User Interface Pattern

Sergej Chodarev
Department of Computers and Informatics
Technical University of Košice, Slovakia
Email: sergej.chodarev@tuke.sk

***Abstract*—Interactive user interfaces include commands that allow to manipulate data or other objects. While in most cases these commands are atomic, it is possible to support composition of simple command elements into complex commands directly during the user interaction. In this case commands have properties of a language. Prominent example of such interface is the Vim text editor. In this paper properties of this user interfaces design pattern are explored and compared with usual interfaces. The ways to integrate this approach with modern interfaces are also proposed.**

I. Introduction

Todays user interfaces (UI) are predominantly oriented on beginners and people that use an interface only sporadically, therefore they need to be able quickly understand it and use it to achieve their needs. For this reason ability to learn an interface quickly – learnability – is the most emphasized property of UI. On the other hand, there are applications that are used by professionals on a day-by-day basis. These applications would benefit from initial learning that would allow to use them more efficiently.

One of the ways how to provide efficient and powerful methods to use computer programs is to use a language-based interaction instead of the see-and-point interaction (see for example the Anti-Mac concept by Gentner and Nielsen [1]). The language does not need to be a natural language that is hard to analyze and ambiguous. Instead, a specially designed domain-specific language [2] can be used, that is both understandable by humans and easy to process by computers.

The goal of this paper is to explore an approach for designing interactive user interfaces with language properties. Let us call this approach *the commands composition pattern*. The essence of the pattern is the following:

Provide a way for a user to compose basic command elements into complex commands directly during interaction. Command elements describe an action that need to be performed and an object to apply the action.

Command elements should be very short, usually they have a form of key presses. They can be seen as an advanced variant of keyboard shortcuts that are combined together to achieve some action.

Main example of the pattern is the Vim text editor. Thus it is used to demonstrate the pattern in Section II. The main contribution of the paper is description of the pattern that is divided into three parts:

1) explanation of the problem the pattern can solve (Section III),
2) description of the solution with possibilities of its implementation (Section IV) and its alternative form based on separate selection and editing steps (Section V),
3) comparison with other approaches (Section VI).

II. Example: Vim Text Editor

The Vim[1] text editor and Vi, as its predecessor, are essential parts of the Unix operating system and its derivatives. Although, the editor is well known for usability problems connected with its modes-based UI [3], the unique principles of human-computer interaction laid in its core can be used as an inspiration for future interfaces.

The Vim editor commands have an interesting property of composability that is well-known among experienced Vim users. For example see the comprehensive StackOverflow answer [4] or blog post explaining the use of this property [5]. Comparison with other editors' approaches is also presented by Kozlowski [6].

In contrast to most other text editors, Vim starts in a *normal* mode where keyboard keys correspond to editing or movement commands. To actually insert some text, a user needs to switch into *insert* mode. There is also a *visual* mode for selecting text fragments and *command-line* mode for writing longer commands.

Vim has a number of motion commands that allow to change position of the cursor relative to the current position. For example `l` moves to the next character and `w` (word) to the beginning of the next word. It also has simple editing commands like `x` which deletes a character after cursor or `s` (substitute) which replaces character after cursor with a new text (it switches to the *insert mode* to allow entering the replacement).

Vim, however, also has more powerful commands which allow more precise editing. For example `d` deletes specified fragment of text. To specify the fragment, a movement command can be used. For example `dl` deletes character after cursor (similar to `x` command) while `dw` deletes text to the beginning of next word.

In addition to the movement commands, text fragments can be specified using *text objects*. For example, command `daw` deletes a word under cursor. In contrast to using movement command (`dw`) this would work even if the cursor is positioned in the middle of the word. Vim provides a number of built-in text objects like words, sentences, paragraphs, text fragments enclosed in parentheses, or XML tags. Additional text objects

[1] Available at http://www.vim.org/

978-1-4673-9868-8/15/$31.00 ©2015 IEEE

are provided by plug-ins, including objects corresponding to programming language syntax constructs (for example program blocks or LaTeX environments).

Composed commands allow to perform complex operations as a single task, thus avoiding repetition. For example, with the *surround.vim* plug-in[2] one can change XML element around a text using a single command and without the need to separately change start and end tags and to move cursor between them. So if there is such text (pipe sign "`|`" marks cursor position):

```
<em>important| text</em>
```

command `cst<strong>` (change surrounding tag) would change `<em>` to `<strong>`:

```
<strong>important| text</strong>
```

In addition, this command can be undone as a whole using standard *undo* command. What is important, such composed commands are abstract and therefore can be easily applied and repeated in different context. This is done by pressing "`.`" key.

III. Problem

The commands composition pattern is suitable for application that allows to execute a set of similar operations on different ranges of elementary objects that form a hierarchy of high-level objects without explicit representation.

For example in text editor there is a hierarchy of paragraphs consisting of sentences consisting of words and at the lowest level there are characters. Text editor, however, directly displays only characters, while words, sentences and paragraphs are present only as sequences of characters recognizable by a user as well as an application based on a well-known grammar. Graphical objects have a similar hierarchy, for example a polygon consists of several lines.

If an application allows to operate with content that has such implicit hierarchical structure, it becomes ambiguous what level of the hierarchy should user commands affect. It is caused by the fact that in a visual representation of the content only lowest level of the hierarchy is displayed explicitly and other levels are present only implicitly as collections of lower level elements.

For this reason, if a user points to some element in the visual representation of the content and invokes some operation, it is not clear to which level of hierarchy the operation should be applied. For example if a user presses *Delete* key, should an application delete current character, whole word, or a sentence.

IV. Solution

The solution is to allow a user to enter commands by composing operations with types of objects they operate on. This means that instead of providing specialized commands for different kinds of objects, application should provide a language for expressing commands. The language should allow to specify an operation that needs to be executed and an object or range of objects that would be affected by the operation.

[2]Available at https://github.com/tpope/vim-surround

Specification of operation and objects may by itself be composed. For example it may allow to specify some modifiers that further refine behaviour of the command or the type of object. For example, Vim text objects have either `i` or `a` prefix for inner (excluding delimiters) and outer (including delimiters) versions of the object. Command can also have a number prefix specifying how many object instances it should affect.

Realization of the pattern requires to choose a way of invoking commands using a sequence of command elements instead of a single action. The elements may have a form of key presses, clicks, etc. The sequence should be displayed while it is entered. This allows a user to see partially entered command and cancel it in case of errors.

In case of applications that are not intended to enter text, command elements can be inserted using keys corresponding to letters. In text editors, where these keys are already used to enter text, other solution should be found. In Vim it is solved by introduction of separate *normal mode* in contrast to *insert mode* where the keys change their interpretation.

Other option is to use modifier keys like *Ctrl* or *Alt* to differentiate command elements from text characters. In contrast to classical keyboard shortcuts consisting of a character key pressed together with one or more modifiers, we would get a sequence of such key presses. This type of shortcut sequences (also called shortcut chords) are used in editors like Emacs or Microsoft Visual Studio, however, with different semantics.

Modifiers itself can be used as command elements to implement a simple form of the pattern. For example a lot of common text editors use a *Ctrl* modifier to change edited object from character to word and *Shift* changes movement operation to selection.

Therefore it is easy to integrate a Vim-like command composition into a modern UI as a special type of keyboard shortcuts that an application may support in addition to conventional user interface. Pressing a key with modifier would not cause immediate invocation of some operation, but instead it would switch the user to command composition mode where he would be able to complete the started command.

It is important to visually indicate the switch by displaying the entered key combination and other command elements when they are entered. This can be done in periphery of the window, for example in status line (see Fig. 1), or in a primary area using some pop-up window. Peripheral version does not cover part of the main working area, but may require some highlighting to call user attention. Pop-up, on the other end, allows to display more needed information, for example contextual help that would improve learnability of the command composition based interface.

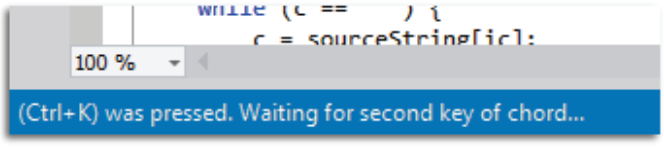

Fig. 1. Prompt for second key of the command in status line of the Microsoft Visual Studio

V. Variation: Select-then-edit Model

An alternative implementation of the described pattern is what we call *select-then-edit* interaction model. It is a model of operation where a user first selects an object or a collection of objects and then invokes an editing operation on them. If no selection is present then editing commands would operate on the most elementary object under cursor. For example if a user presses *Delete* key in a text editor, then a character after cursor would be deleted. If, however, a user would first select a whole sentence and then press *Delete*, then the sentence would be deleted.

This model is used in most of the current graphical user interfaces and can be viewed as a special case of the commands composition pattern. Commands are there composed from two parts: selection subcommand and editing subcommand.

Main advantage of such approach is the fact that selection is usually immediately visible and therefore the process of command construction is more visual[3]. User first selects objects and during this process he has a visual feedback about the selected range, then he executes a command on the selection he sees.

Despite the advantages, this approach requires powerful selection operations to be comparable with the Vim-like approach. Usually, selection can be started at current cursor position and then gradually extended using movement commands with pressed *Shift* key or using a mouse. Even if an editor provides powerful enough movement commands this still does not provide alternative to Vim text objects.

In addition, compared to Vim-like solution, this approach also does not have command repeatability property. This limitation, however, can be overcome using macro recording.

To unleash full power of the command composition pattern it is important to extend currently widespread implementation with richer vocabulary. This requires addition of selection commands for different text objects in addition to usual selection based on movement.

For example the *Extend Selection (Ctrl+W)* command in the JetBrains IntelliJ Idea[4] is a step in this direction. It allows to select a syntactically relevant part of text around current cursor position. Each press of the key combination extends the selection to the larger element of the text, thus moving to the higher level of hierarchy.

Extend Selection command, however, still requires repeated key presses to select desired fragment. Specialized selection commands would overcome this limitation and in addition they would allow selection of non-continuous fragments, for example XML opening and closing tags without enclosed text. It is important to use consistent keyboard shortcuts for selection commands and same modifier keys if it is possible.

VI. Comparison with other Solutions

The most trivial alternative to the commands composition pattern is to simply ignore the problem described in Section III and apply operations only on elementary objects. Therefore, in the *Delete* key press in an editor would cause deletion of one character.

This solution may be suitable for casual users as it is very easy to learn and at the same time allows users to operate on higher levels by repeating commands or by manually selecting corresponding sequence of low-level elements. This approach of work is, however, much less efficient then direct manipulation with high-level objects and this becomes problematic for professional users of an application. For this reason specialized commands or shortcuts for most common objects are often added. For example *Ctrl+Delete* to delete whole word. This leads to introduction of separate ad-hoc commands for different combinations of operations and objects. These commands usually can be accessed using a menu and keyboard shortcuts.

If the number of operations is large, menus would become cluttered and require nested submenus. Keyboard shortcuts, on the other hand, would suffer from limited number of keys and thus require the use of different modifiers and combinations of modifiers. Choosing intuitive and easily remembered key combination for each operation would become a very hard task. The result may become a collection of deep and complex menus and ad-hoc shortcut key combinations that are hard to remember. The problem can be mitigating by introducing search functionality for commands, but this would not resolve it fully.

The main advantage of the commands composition pattern is, therefore, the ability to provide large number of commands that can by described by a small set of elements. This makes it simpler to learn commands compared to the situation where every command would have a separate invocation method. A user needs to learn only command elements and rules of their composition (command language grammar).

On the other hand, such approach of interaction is not usual for most of users and requires special learning. This approach also suffers from low discoverability. For this reason it is suitable only for professional users who can afford investment of their time for improved productivity.

VII. Related Work

Description of common solutions in form of patterns inspired by the work of Alexander [7] become quite wide-spread in computer science. It is used in different areas from object-oriented design [8] to XML to annotations mapping [9]. Design patterns for user interfaces and human-computer interaction are presented in several pattern collections, including a book *Designing Interfaces* by Tidwell [10] or a more narrowly focused work by Van Duyne et al. on design of web sites [11]. Overview of several user interface design patterns sources can be found in the paper of Kruschitz and Hitz [12]. In contrast to them, this paper discusses only a single pattern, not the whole collection.

Language properties of user interfaces are covered in the work of Bačíková et al. [13], which considers any definition of user interface (for example in a form-driven way [14]) to be also a definition of domain-specific language. For example, a form defines a language for expressing data that are entered

[3]It is not a coincidence that a mode allowing such interaction in Vim is called *visual*.

[4]Extend Selection command is documented at https://www.jetbrains.com/idea/help/selecting-text-in-the-editor.html#d1701627e179

into it. The authors also consider how language and domain aspects of UI should be included in usability evaluation [15]. In this paper a narrower view on languages is used and it is concerned only with commands for manipulating some content that have some non-trivial grammar and are used directly during user interaction with an application.

Experiments of van Nimwegen et al. [16], [17] show the role of user interface on user performance and problem solving strategies. The results suggest that user interfaces that support internalization of information lead to better users performance compared to interfaces that try to externalize information by assisting users.

Experiment of Chen et al. is also devoted to comparing different user interface styles [18]. In this case – graphical and textual ones. The results confirmed that graphical user interface lead to better performance of novice users, in the case of experts, however, the difference was not significant.

An attempt to compare influence of command-line tools and Vim editor with graphical integrated development environment to the learning of programming was done by Dillon et al. [19]. Part of the respondents, however, was not trained in using Vim, so they failed to write any code in the editor. This fact degrades the results of the comparison.

The other area that is significant for the topic of the paper and is only briefly covered there is the help that an application should provide its users to learn more efficient interaction models and therefore become expert users. This topic is covered extensively by Cockburn et al. [20].

VIII. Conclusion

This work presents an attempt to discuss the user interaction pattern that was mostly overlooked. To fully evaluate the potential of the pattern it is, however, needed to carry empirical research on real users. Both comparison of an interface based on commands composition with canonical interface and comparison of different variants of the commands composition (Vim-like and select-then-edit) are interesting topics of future research. Evaluation of longterm advantages and disadvantages of interface intended for experts is, however, much harder than evaluation of interfaces for novices.

The comparison can be done on the existing examples of the pattern implementation. Therefore another important task is collection and analysis of such real-world examples.

More objective comparison and evaluation, however, would require development of prototype user interfaces (either standalone or as extensions to existing software) that would realize needed interaction models. This approach would decrease accidental factors that may impair validity of acquired results. In addition it would allow to design and evaluate improved variants of the pattern. The main area for such improvement is development of the ways how to lower learning costs required to fully master such user interface.

Acknowledgment

This work was supported by project VEGA 1/0341/13 "Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication".

References

[1] D. Gentner and J. Nielsen, "The Anti-Mac Interface," *Commun. ACM*, vol. 39, no. 8, pp. 70–82, Aug. 1996. [Online]. Available: http://www.nngroup.com/articles/anti-mac-interface/

[2] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[3] I. S. MacKenzie, *Human-computer interaction: An empirical research perspective*. Morgan Kaufmann, 2013.

[4] J. Dennis, "Your problem with vim is that you don't grok vi," StackOverflow answer, Dec. 2011. [Online]. Available: http://stackoverflow.com/a/1220118

[5] J. Carroll, "Vim text objects: The definitive guide," Oct. 2011. [Online]. Available: http://blog.carbonfive.com/2011/10/17/vim-text-objects-the-definitive-guide/

[6] M. Kozlowski, "Why Atom can't replace Vim," Mar. 2014. [Online]. Available: https://medium.com/@mkozlows/why-atom-cant-replace-vim-433852f4b4d1

[7] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] M. Nosál and J. Porubän, "Xml to annotations mapping definition with patterns," *Computer Science and Information Systems*, vol. 11, no. 4, pp. 1455–1477, 2014.

[10] J. Tidwell, *Designing interfaces*. O'Reilly, 2010.

[11] D. K. Van Duyne, J. A. Landay, and J. I. Hong, *The design of sites: Patterns for creating winning web sites*. Prentice Hall Professional, 2007.

[12] C. Kruschitz and M. Hitz, "Analyzing the HCI Design Pattern Variety," in *AsianPLoP '10: Proceedings of the 1st Asian Conference on Pattern Languages of Programs*. New York, NY, USA: ACM, 2010.

[13] M. Bačíková, J. Porubän, and D. Lakatoš, "Defining domain language of graphical user interfaces," in *SLATE 2013: 2nd Symposium on Languages, Applications and Technologies*, vol. 29, 2013, pp. 187–202.

[14] S. Ristić, S. Aleksić, I. Luković, and J. Banović, "Form-driven application development," *Acta Electrotechnica et Informatica*, vol. 12, no. 1, pp. 9–16, 2012.

[15] M. Bačíková and J. Porubän, "Domain usability, user's perception," in *Human-Computer Systems Interaction: Backgrounds and Applications 3*, ser. Advances in Intelligent Systems and Computing, Z. S. Hippe, J. L. Kulikowski, T. Mroczek, and J. Wtorek, Eds. Springer International Publishing, 2014, vol. 300, pp. 15–26.

[16] C. Van Nimwegen, H. Van Oostendorp, H. Tabachneck-Schijf *et al.*, "The role of interface style in planning during problem solving," in *The 27th Annual Cognitive Science Conference*, 2005, pp. 2771–2776.

[17] C. van Nimwegen, H. van Oostendorp, D. Burgos, and R. Koper, "Does an interface with less assistance provoke more thoughtful behavior?" in *ICLS'06: 7th International Conference on Learning Sciences*. International Society of the Learning Sciences, 2006, pp. 785–791.

[18] J.-W. Chen and J. Zhang, "Comparing text-based and graphic user interfaces for novice and expert users," in *AMIA Annual Symposium Proceedings*, vol. 2007. American Medical Informatics Association, 2007, p. 125.

[19] E. Dillon, M. Anderson, and M. Brown, "Comparing Mental Models of Novice Programmers when Using Visual and Command Line Environments," in *ACM-SE'12: Proceedings of the 50th Annual Southeast Regional Conference*. New York, NY, USA: ACM, 2012, pp. 142–147.

[20] A. Cockburn, C. Gutwin, J. Scarr, and S. Malacria, "Supporting Novice to Expert Transitions in User Interfaces," *ACM Computing Surveys*, vol. 47, no. 2, Nov. 2014.