VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Smarter Cut-and-Paste for Programming Text Editors

Glen Wallace, Robert Biddle and Ewan Tempero

Technical Report CS-TR-00/4
August 2000

Abstract

The process of creating software involves many different tools, but the text editor is still one of the most important. Moreover, one of its basic facilities, cut-and-paste, still plays a critical role in enabling simple reuse. In this paper we explore how to improve cut-and-paste within text editors used for programming. We describe how programmers use cut-and-paste for reuse, and suggest the basic underlying principles. We then report on a prototype tool we developed to explore a set of techniques to provide better support for cut-and-paste.

Publishing Information

A later version of this paper was published in the Proceedings of the Australasian User Interface Conference 2001, Gold Coast, Australia, IEEE Computer Society Press.

Author Information

Glen Wallace was a BSc(Hons) student at Victoria University of Wellington during the work described in this paaper, Robert Biddle and Ewan Tempero are lecturers in Computer Science, also at Victoria University of Wellington, New Zealand.

Smarter Cut-and-Paste for Programming Text Editors

Glen Wallace, Robert Biddle and Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
New Zealand
gwallace@the.net.nz {Robert.Biddle|Ewan.Tempero}@mcs.vuw.ac.nz

Abstract

The process of creating software involves many different tools, but the text editor is still one of the most important. Moreover, one of its basic facilities, cut-and-paste, still plays a critical role in enabling simple reuse. In this paper we explore how to improve cut-and-paste within text editors used for programming. We describe how programmers use cut-and-paste for reuse, and suggest the basic underlying principles. We then report on a prototype tool we developed to explore a set of techniques to provide better support for cut-and-paste.

1 Introduction

Software development involves many different activities, and these activities have led to a variety of tools to support them. One of the early tools is still widely used today: the text editor. Text editors have a very general interface, and provide primitive but general support for creating programs in traditional textual programming languages. Our general research interest is in code reuse, and text editors provide support for reuse through mechanisms for textual cut-and-paste. This support is really very primitive, but it is heavily relied on by programmers to reuse old code and adapt it to new circumstances. In this paper, we describe a project that explored how text editors might better support cut-and-paste in order to better support code reuse.

Programming is a complex task, and even the processes involved when the programmer interacts with a text editor are quite complicated. In particular, programmers work through text editors to explore and develop programs in a creative and dynamic way that does not follow predetermined plans. For these reasons, we choose to base our exploration on an analysis of what programmers actually do when using text editors. The first step in our exploration was to describe the ways in which programmers used cut-and-paste to reuse and adapt code. We then tried to identify the key concepts involved in these activities, to form the basis for higher-level support within a text editor. The main work of the project was then to develop a prototype text editor based on these concepts.

Our hope is that these efforts will lead to improved text editors. Text editing is an important user interface element, and editors are important because users spend significant time and effort working within them, even in single sessions. Understanding and improving text editors is therefore a worthwhile objective.

Beyond editors, our aim is to better understand how programming languages and programs should support reusability. Our previous work has included a conceptual model for programming language support for reuse [1]. Our original intention in developing that model was to better understand programming language design. However, the design of programming languages impacts the ways in which programs are constructed, and especially affects how parts of a program are put together.

The rest of this paper is structured as follows. Section 2 describes some key ways in which programmers use the primitive facilities of text editors to reuse and adapt code, and identifies some underlying concepts that emerge. Section 3 describes Machete, our prototype tool that adds visualisation to better support text editor cut-and-paste. We then discuss our experience in section 4, and finally present our conclusions in section 5.

2 Using Text-Editors to Cut-and-Paste Programs

The basic ideas of cut-and-paste are now very widespread, familiar, and reasonably consistent. Despite the common term "cut-and-paste", there are typically three operations: cut, copy, and paste. The user may *select* a *source region*, and invokes either a *cut* or *copy* operation. In both operations, the selected text is copied to a *buffer*; cut also deletes the text in the source region. The user may also indicate a *destination location*, and invoke a *paste* operation, which copies the text from the buffer to the indicated location.

The first step of our study was to form an initial catalogue of the ways in which cut and paste is used within text editors for writing programs. Our procedure was to focus on one particular text-editor, on one particular programming language, and to gather information by observation of ourselves and other programmers. We used the Emacs text editor [9], and the Java programming language [4]. All the programmers involved were experienced with these, and we regard the usage of each as sufficiently typical of text editor usage and programming use of a text editor. In particular, although Emacs allows sophisticated and extensive customisation, the usage of Emacs we considered did not rely on this.

We recorded interesting usage as "use cases". A use case is a description of a typical interaction in a system [5], and the term is widely used in the field of object-oriented design to attempt to describe the functionality a system should provide. We identified a number of use cases, and summarise them below as concerning principally blocks of code, variables, methods, or refactoring. It might seem that only the first category, blocks of code, relates to cut-and-paste. However, we observed that the other categories are also related, because cut-and-paste typically involves extensive browsing and reviewing of code, and extensive modification and customisation.

2.1 Manipulating Blocks

Manipulation of blocks of code includes various kinds of code structure. We use the term *fragments* to refer to program statements, lists of statements or blocks, being a number of statements enclosed in brackets. The term *expression* we use to refer to a valid expression including any language construct that can be evaluated to produce a value, such as arithmetic operations and method invocations. So, the use cases include:

- Insert, remove, or reorder code fragments, and highlight any changed dependencies between the fragment and the surrounding context.
- Insert, remove, or replace expressions, and highlight any inconsistencies or changed dependencies.
- Reusing code structures.

  This common and important kind of code reuse involves copying a block of code, where whole block of code is not being reused, but rather the underlying structure of the code. Accordingly, the user then replaces parts of the block with their own code as appropriate. This form of reuse is common when using library or framework code, where example code is taken from the documentation or tutorials. The important points to consider are:

  – Which control structures are of interest and which should be discarded.
  – Which variables form part of the control system in the code. Which variables should be replaced by variables in the destination context, and which variables are irrelevant and should be removed.
  – Which methods contribute to the control of the structure, which methods contribute to the algorithm being reused, and which methods are simply padding for the purpose of the example.

Our earlier work on code reuse developed a conceptual model for the ways in which languages allow code components to be invoked at run time from different contexts [1]. Many key issues concern *dependencies* between the component and the contexts. For example, a context may depend on a component taking certain parameters, or a component may depend on the context setting the value of some global variable. In general, dependencies limit reusability, but we classified a variety of dependencies to better explain the exact nature of their impact.

In text editor cut-and-paste, dependencies also play a key role. There are dependencies between a selected block of code and the surrounding text, and these may be affected by any cut or paste. In the processes of reusing code by cut-and-paste, the programmer must consider the dependencies, and address these where necessary. Much of this work involves manipulating variable usage and method calls, as discussed below.

2.2 Manipulating Variables

The category involving manipulation of variables includes a number of use-cases, but we must remember that there is an area of overlap with the manipulation of methods, because of the object-oriented nature of the language: where the variable refer to objects, they are used by invoking their methods. The identified use-cases are summarised below.

- Display the variables in scope at the indicated point.
- Display the variables that are referenced in a selection but undeclared there.
- Display the shadowed variables, which would be usable but for others with the same name.
- Display a variable's type and origin.
- Display a variable's places of usage.
- Display all the methods and fields that are invoked for an object variable.

- Determine the data flow through variables in the code: which variables affect any given variable, and which variables any given variable affects.
- Change the name of a variable.
- Introduce a new variable.
- Remove an unused variable.
- Remove all usage of a variable.
- Shift variables from local within a method, to a field; or vice-versa.
- Change the declaration attributes of a variable, and highlight introduced inconsistencies.

2.3 Manipulating Method Calls

There are numerous ways method calls may be manipulated, as shown below. While we are assuming use of Java, an object-oriented language, many of these cases would also apply to procedural languages as well.

- Show the methods in scope, identifying:
  - Inherited methods, and the relevant superclass.
  - Overriding methods, and the relevant superclasses.
  - Any interfaces the method takes part in implementing.
  - Static methods.
  - Instance methods.
- Show a method's signature, including parameter and return types.
- Show variables and method return values that are used as parameters to method invocations.
- Show variables and method parameters that take values from method return values.
- Change the name of a method.
- Change the name of a method applied to a particular object, or type of object.
- Remove all usage of a method.
- Modify the parameters of a method, and highlight introduced inconsistencies, of changing their order, adding parameters, removing parameters, or changing parameter types.

2.4 Refactoring

Refactoring is the term usually applied to make the code more maintainable or reusable through improvement to its internal structure [3]. Refactoring can involve either code level changes or architectural changes that effect the global structures and abstraction of the program.

Code level changes can include reorganising expressions, moving common code to or from helper methods, and restructuring control flow statements. Architectural level can involve extracting a composition or inheritance relationships, moving functionality within inheritance hierarchies, and partitioning functionality differently within classes.

Accomplishing refactoring simply using a text editor will typically involve sequence of low level cut, copy and paste operations. However, the essence of the modification at a higher level is to change the structure of the code without changing the functionality at all. This means that these operations are quite constrained, and this allows specific higher level support for refactoring. Because the modifications involved in restructuring do allow tighter constraints and better high-level support, we chose to focus initially on the less constrained use-cases discussed in the previous sections.

2.5 Towards Better Support for Cut-and-Paste for Programs

The first step in our project was to describe the use-cases where cut-and-paste was used in programming. The problem is that it is a very low level concept, and even good text editor support leaves the user with a high mental overhead of determining the implications of steps, and keeping track. To conclude this step in our project, we analysed the user tasks involved in the use-cases, and tried to identify higher level concepts to better support the user.

It seemed that the majority of the use cases described involved a relatively small number of concepts:

- Displaying variable declarations and method definitions.
- Showing variable scoping, or visibility.
- Access control, determining whether access is permitted.
- The inheritance hierarchy.
- Type inference and type compatibility checking.
- Determining dependencies, seeing how a variable or method affects other code.
- Determination of where and how variables and methods are used in the code. For variables, this involves the fields and methods referred to through each object variable. For methods, this involves the parameter types, and return type.

Implicit in many cases is the underlying idea of a *navigation* around the code. We need to index the code in several ways, move from definitions to usages or back, and see dependencies between elements.

In the past, the recognition that programs follow highly structured syntactic forms have been used to support the argument that code should not be treated as mere text [8].

Figure 1. The Machete interface provides a split view for navigation, the tree display on the left and the code display on the right. (The pane widths have been reduced for this figure.) The tree view shows definitions of names, organised according to the scopes of the names.

Construction tools that make use of this language knowledge were termed "syntax-directed editors". These tools failed to gain widespread user acceptance mainly due to the way in which they were perceived to restrict the way development took place. In design to better support cut-and-paste by using syntactic structure, we must be cautious to avoid constraining development in the same way syntax directed editors were thought to do.

3 Machete: a Prototype Tool

In this section, we describe Machete, our prototype tool for exploring support for code reuse. The main goal of our prototype was to exploit our analysis of the issues that programmers face when using cut-and-paste. Our analysis suggested that the main issue faced by programmers is understanding the consequences of their action. Specifically, understanding the dependencies that source region has on its context that would be affected by pasting it in the new context. Accordingly, we were interested in a user interface that provided a visualisation of the code fragment and its context that would make it easy for the programmer to reach this understanding.

3.1 Prototype Design

To meet our goal, our prototype did not actually have to provide editor support — in fact it did not even have to support the cut operation. All that was really needed was to allow the user to select a fragment of code and to provide information about that fragment of code.

To provide the information of a selected fragment of code, our prototype had to be able to analyse that fragment. Such analysis would require some level of parsing of the fragment. This raised the issue of what selection of code would be allowed. Text editor users are used to being able to select arbitrary pieces of text, but such selections could easily result in code fragments that were unparsable. Dealing with such fragments is a similar problem to the problem of compilers providing good error reporting for incorrect code[2], however this was not our interest. Furthermore, our observation of programmers suggested that for the most part programmers who are selecting a piece of code with the intention of copying it to use somewhere else will typically choose a segment that falls on natural syntactic boundaries. Accordingly, we choose to restrict selection to syntactically complete fragments of code, that is groups of expressions and/or statements. While we want to restrict the selection, doing so by forcing the programmer to be careful would be counter-productive. This means we need a mechanism that allows a programmer to easily make allowable selections. This "guided" selection places restrictions on the kind of code that can be reused, and raises issues as to how allowable selections are indicated in the user interface.

As we have discussed above, the visualisation provided by our tool had to make the dependencies clear. At the level of reuse that we are interested in, dependencies are mainly indicated by definition and use of names, which is what our tool concentrates on. A common issue that a programmer must address in relation to a fragment of code under consideration for reuse is to determine where a name in that fragment is defined. In a standard text editor, answering that question will typically involve traversing the file to find the definition. There are two tasks being performed in such action — navigation and identification — and both must be supported. We chose to use a split view to support navigation, with one pane (the tree display) providing an indexing structure on the code loaded into Machete, while the other pane (the code display) shows the actual code. We also make extensive use of colour and icons to support identification. (See figure 1).

3.2 Selection

Having made the decision to restrict the fragments of code a programmer could select for copying, we had to then provide a simple mechanism to allow the programmer to easily make such a selection. The mechanism had to be predictable in its behaviour, easy to use, and provide good feedback. In particular, it had to be a clearly superior mechanism to use than the standard cut-and-paste mechanism of editors such as Emacs.

The mechanism we chose was to have the user indicate a start point and then expanding or contracting the selection about this point. The granularity of the change in selection is to increase (or decrease) to the next "enclosing" syntactically complete block. Thus the first selection will be of a variable, which is then increased to include the expression the variable is in, then the statement, then the block, and so on (see figure 2). This is similar to behaviour that has been available in some editors for several years, where a single click sets the starting point, a double click selects a "word", a triple click a "line", and so on. Our selection mechanism is smarter with respect to the syntax of the language.

The change in selection is achieved through the mouse. If a selection exists, a left click within the selection will cause it to expand to the next step, while a right click will cause it to contract to the previous step. If there are no remaining steps to expand or contract to, then the operation will do nothing. Clicking the centre button at anytime, or the left button in a region outside the current selection, will cause the selection to be cleared. This allows the programmer to quickly cycle through the allowable selections with little cognitive overhead. Notice that this mechanism provides no explicit information regarding what the next

Figure 2. Code is selected by choosing a starting point and expanding the selection from that point. This shows the sequence of expansions starting with the local variable `localDecl`. On the initial selection (a), the definition of `localDecl` has been coloured (bold red). With each left click, the selection expands to the next enclosing syntactic fragment, a method name (b), method call with no arguments (c), `if` condition (d), `if` statement (e), and `if` statement block (f). As new names are included in the selection, other parts of the code are coloured (for example the field `outScopeDeclAll` is coloured in the `if` statement following the selected fragment in (e), and the local variable `isLocal` is coloured in (f)). Figure 1 is the result after 2 more left clicks.

selection would be in the expansion/contraction sequence, however, provided the implementation is quick enough, the mechanism is simple enough to allow the programmer to experiment until the desired selection is determined.

The current selection is indicated using the traditional mechanism of changing the colour of the background. We considered more sophisticated feedback, such as different fonts or graphical overlays, but decided to stay with what the programmer would be familiar with for this prototype.

3.3 Navigation

Navigation is provided by an indexing structure based on the definition of names. This is a tree structure organised according to the nesting of scopes, and with labelled internal nodes corresponding to different collections of definitions and leaves corresponding to the definitions themselves. In this way, the scope of definitions is clear. The tree nodes can be expanded or collapsed by direct user control, but will also be changed by the current selection (see below).

The programmer may click the left mouse button on a node in the tree display and this will result in the code display scrolling to place the contents of that node in the centre of the code display. This allows the user to quickly locate the declarations. Using the right mouse button will achieve the same effect, but also the declaration in question will be selected, which will cause all the uses of that variable to be highlighted.

3.4 Identification

Different icons in the tree display are used to show the different kinds of declarations, as shown in left pane of the user interface shown in figure 1. Groups of similar declarations (for example, fields, methods, method arguments) use a "container" icon, with the different groups indicated by different colours. Declarations of variable-like names (such as fields, arguments, local variables) are shown with right-slanted rectangles, again using colour to differentiate each kind. Non variable-like names (such as method names) use left-slanted rectangles. Class-like names use circular icons, return values are shown by the backwards arrow, and blocks are shown by a horizontal rectangle. Blocks relating to statements are specialised icons (for example, `if` and `for` statements). This approach can be further extended to have the icons convey more information in the form of accessibility and variability information.

In both the tree and code displays declarations and usages of interesting variables are highlighted. The colour that the usage or declaration is highlighted indicates what it is and how it relates to the current selection. There are four cases for colouring:

- Declarations of variables that are used within the selection, but declared outside are marked in bold red.
- Usages outside the selection of variables used (but not declared) within the selection are marked with brick red.
- Declarations that occur within the selection are marked with bold blue.
- Usages within the selection of variables declared within the selection are marked in purple.

For example, in figure 1, the declaration of the variable `var` is coloured bold red, because it is used within the selection. All uses of it outside the selection are coloured brick red. The variable `localDecl` is coloured blue on its declaration, and purple whenever it is used.

The selection mechanism is tied to the tree display. As the selection expands and contracts the variable declarations in the tree display and the variable declarations and uses in the code display will be updated so that their highlighted state will reflect the dependencies between the currently selected fragment and the surrounding context. The relevant tree nodes will be expanded to show the definitions of the various names used within the selection, including any fields. The names so used will also be coloured in the tree display to make them easily identifiable. This means that all the definitions affected by the selection are immediately visible. The navigation mechanism can then be used to explore the consequences of moving the selection to a new context.

3.5 Implementation details

Machete was implemented in Java. It uses JavaCC [6], a Java compiler compiler, and JTB [7], a syntax tree builder, to parse the input file and construct an abstract syntax tree, that is the basis for both the selection mechanism and the tree display. It uses the `java.swing` package for the basic user interface components, and `java.swing.text` for the text display and the selection mechanism.

4 Discussion

Text editor support for programming is a large subject, and this project was much more limited in scope. However, the prototype did allow us to explore a set of techniques designed to better support cut-and-paste.

We evaluated the prototype principally by using the "cognitive walkthrough" technique [10]. Normally, our practice would be to then proceed to usability testing, but the limited nature of the prototype would have made this difficult. Moreover, we had some difficulties with the Java run-time performance, and our prototype was slower than is necessary for normal use of a text editor.

Our study of the prototype did show that our design did appear to provide better support for the basic elements involved in using cut-and-paste. Compared with an ordinary text editor, it was clear that the prototype significantly increased support for both navigation and detection of dependencies. We found ourselves exploring code by using the new features, using the tree navigation and guided selection, and assessing the highlighted information identified in the code. This suggests that this form of support fosters active exploration and reflection, which is useful just for code understanding, without the reuse focus. In some way the features made the prototype seem faster that we were used to, despite the overall slowness of our prototype implementation. In particular, the guided selection worked well, and we found that as well as using to select blocks, we also used it to identify and explore the structure of the code, while also being able to rapidly review identified details.

Guided selection, declaration and structure navigation, and code identification and highlighting all seem basically sound user interface techniques. Overall, we think this is evidence that text editors can provide better support for cut-and-paste, by using available syntactic structure, but without going all the way to the restrictions of syntax-directed editing. In fact, it seemed that a little went a long way. Even small improvements beyond the typically primitive text editor capabilities might might make code reuse significantly easier. Future work will need to explore more full support for smarter cut-and-paste in a working text editor.

5 Conclusion

This paper has reported on our exploration of smarter cut-and-paste in text editors used for programming. We began by describing the ways in which programmers use cut-and-paste, and by identifying the key underlying concepts. We then used this analysis to drive the design of a prototype tool to provide better cut-and-paste support. The tool was limited in scope, but appeared to show the ideas had merit. It does seem possible to better support cut-and-paste through guided selection, navigation by declarative structure, and by highlighting identified details within the code.

Cut-and-paste is a primitive interface capability, but plays a surprisingly large role in how software developers achieve code reuse in practical program development. There are many higher level approaches to supporting code reuse, and these may have advantages relating to correctness, scale, and modularity that cut-and-paste can never have. However, there is still a place for cut-and-paste. It is a low-level tool, but it is highly usable, highly versatile, and often the most expedient way to reuse existing code. Cut-and-paste is worth taking more seriously, and worth better support.

References

[1] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*. IEEE Computer Society Press, April 1996.

[2] Gordon V. Cormack. An LR substring parser for non-correcting syntax error recovery. In Bruce Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pages 161–169, Portland, OR, USA, June 1989. ACM Press.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, USA, June 1999.

[4] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, USA, 1996.

[5] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A use case driven approach*. Addison-Wesley, 1992.

[6] Homepage for JavaCC, Sun Microsystems, Inc. `http://www.suntest.com/JavaCC/`.

[7] Homepage for JTB, Department of Computer Science, University of Purdue. `http://www.cs.purdue.edu/jtb/`.

[8] Amir A. Khwaja and Joseph E. Urban. Syntax-directed editing environments: Issues and features. In Ed Deaton, K. M. George, Hal Berghel, and George Hedrick, editors, *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pages 230–237, Indianapolis, IN, February 1993. ACM Press.

[9] Richard Stallman. *GNU Emacs manual*. Free Software Foundation, Cambridge, USA, 13th, Emacs version 20.3 edition, July 1997.

[10] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. The cognitive walkthrough method: A practitioner's guide. In Jakob Nielsen and Robert L. Mack, editors, *Usability Inspection Methods*, chapter 5, pages 105–140. Wiley, 1994.