Three Public Enemies: Cut, Copy, and Paste

Copy-paste might seem to make life easier, but it often leads to inconsistencies. Giving users the freedom to specify semantic relationships among copied objects can help rectify this crime.

Zoltán Ádám Mann
Budapest University of Technology and Economics

Many software developers know the feeling of desperately debugging a program only to discover after a sleepless night that the error stemmed from copy-pasted code segments that had become inconsistent in subsequent editing. Even worse is the sinking feeling that many more such subtle errors are waiting to be discovered.

The problems arising from copied code are not new, and many researchers have investigated how to automatically find copied code segments.[1-4] An often recommended approach is to *refactor* the copied code—generalize the segments to a meaningful programming language abstraction, such as function, macro, or class, and replace the segment's occurrences with references to that abstraction.[5] However, refactoring is not always the method of choice[6] and is obviously not a solution for other kinds of text editing, which suffer from the same copy-paste deficiencies.

Given the extensive use of copy-paste operations and their tendency to cause inconsistencies, there is clearly a pressing need to rethink current editor programs. One solution is to replace cut, copy, and paste with operations that correspond directly to the intended semantics behind their use. With these operations, the user can specify semantic relationships among copied objects and the editor program can use that information to help in the long-term support of those relationships. It would thus avoid the inconsistencies that currently arise from the use of cut, copy, and paste.

USE SCENARIOS

To understand why cut, copy, and paste can produce errors, it helps to see exactly why people use these operations. There are five typical scenarios, some of which can create inconsistencies:

Cut to delete. Although this is not what its creators intended as the cut function—and most programs do indeed have a separate delete operation—it is nevertheless possible. However, such use does not lead to inconsistencies, so this scenario is relatively harmless.

Cut and paste to move something. Again, there's no potential for inconsistencies, since something is simply moving from one place to another.

Copy and paste to duplicate. The idea is to have exactly the same thing at several places. Typing variable names is a good example of this scenario. Software

0018-9162/06/$20.00 © 2006 IEEE

engineers learn to use more descriptive variable names than *i*, *j*, or *k*, so they might create a variable myActionListenerForTheUpperRightButton, which they might need to access a dozen times. A possible strategy is to copy and paste the variable name.

Copy and paste to create a template. In this case, the aim is to have two similar instances. Continuing the example in the previous scenario, suppose a software engineer has finished the code that handles myActionListenerForTheUpperRightButton, and now he wants to do the same thing with myActionListenerForTheUpperLeftButton. To avoid going mad, he copy-pastes and modifies the code.

Copy and paste without a logical connection. In this case there is no logical relationship between original and replica. For example, when programming a Java class, a software engineer might start the declaration of several methods with public void. To save time, she might want to type this string only once and copy-paste it for the other methods, and change it for those that are, say, public int. This example differs from the previous one on the point of logical connection. In the previous case, the text copies have a logical connection; here they do not. If the software engineer changes the code that handles myActionListenerForTheUpperRightButton, chances are high that she will make analogous changes to the other copy. However, if she changes a method's modifier from public to private, she won't usually need to change the modifier of the other methods in the same class. Consequently, this scenario is less harmful—text copies need not be consistent, so there's no chance of inconsistency problems.

These five basic use scenarios also cover one more interesting scenario: cut and then several pastes, which is really a move operation followed by one or more copies.

WHY THINGS GO WRONG

Inconsistencies arise in the third and fourth use scenarios. The most frequent problem is that the user changes some text copies but forgets to change the others. In the fourth use, forgetting to change the copied text can open another possibility for inconsistency.

This is especially problematic for large, long-lived texts—manuals, specifications, books, and program code, for example—that must be updated multiple times during their life. In such cases, the writer often no longer remembers what additional changes she must make to text to make it internally consistent, or the original writer has moved on and others have inherited the text.

At the root of these errors is a key restriction: The user can't specify the exact semantics of copying. In their present form, cut, copy, and paste help users replicate text, but they don't help with the task of making sure that the copied instances remain consistent. What is the point of having a computer if it can't perform this kind of task?

TOWARD THE IDEAL PROGRAM

The ideal editor program would have two key features. First, it would *not* use a clipboard. The clipboard is problematic because it offers only short-term support for copying text and does not let users specify the semantics behind the copy operations. Second, instead of the traditional cut, copy, and paste operations, the ideal program would provide four operations that correspond to the use scenarios just described: move, copy-identical, copy-and-change, and copy-once.

Forgetting to change the copied text is especially problematic for large, long-lived texts.

The *move* operation is exactly what it implies: The user moves text completely from one place to another with no other effect. Thus, the move operation does not lead to inconsistencies. This contrasts to the currently used combination of cut and paste, in which text remains on a clipboard, thus enabling the user to paste the text in additional locations, which can lead to inconsistencies.

The *copy-identical* operation corresponds to the third use scenario described earlier—copying to have exactly the same thing at several places. But copy-identical would not put the burden of ensuring consistency on the user. Instead, after a copy-identical operation, the editor program would guarantee that the copies remain identical. That is, when the user employs the copy-identical operation, the program memorizes which pieces of the document must remain identical. When the user changes one copy, the program automatically updates the others. The program also tells the user how many other copies it has changed and invites the user to look at the other copies and possibly break the stored relationships.

The *copy-and-change* operation addresses the fourth use scenario—copy text and modify the new instance to obtain something similar, but not identical, to the original. The program isn't as proactive in this case because it can't know how to apply a change in the original to the replica. However, it *can* memorize when the user employs the copy-and-change operation and notify her when it is possible that a change could lead to inconsistencies.

As long as the replica remains unchanged after copying, the program should mark it with a suitable visual reminder to the user that she wanted to change it. When the user does change the replica, the program would remove the reminder. When the user later changes the replica again, the program should make it possible to navigate to the original and change it if the user so desires, but it would not signal her that she *must* change

the original. On the other hand, if a user changes the original, the program should tell her which replicas she might have to change as well. Again, the program should offer the user the chance to investigate and potentially break the stored relationships.

Finally, the *copy-once* operation addresses the fifth use scenario—copy text and possibly change it without implicitly wanting to define a logical relationship between original and replica. This is similar to the normal copy-paste, in which the user copies text but the program does not store any relation between the original and the replica and will not try to help the user avoid inconsistency problems. This operation is also suitable for people who have grown up with copy and paste and do not want to become familiar with new semantics, as well as for those who become annoyed when computers try to be more intelligent than their users.

In addition to these four operations, the program should provide a way to mark text for the next operation. In contrast to copying something to the clipboard, marking has no remaining side effects beyond the next operation. This is not possible with traditional copy-paste because there is no way to remove something from the clipboard.

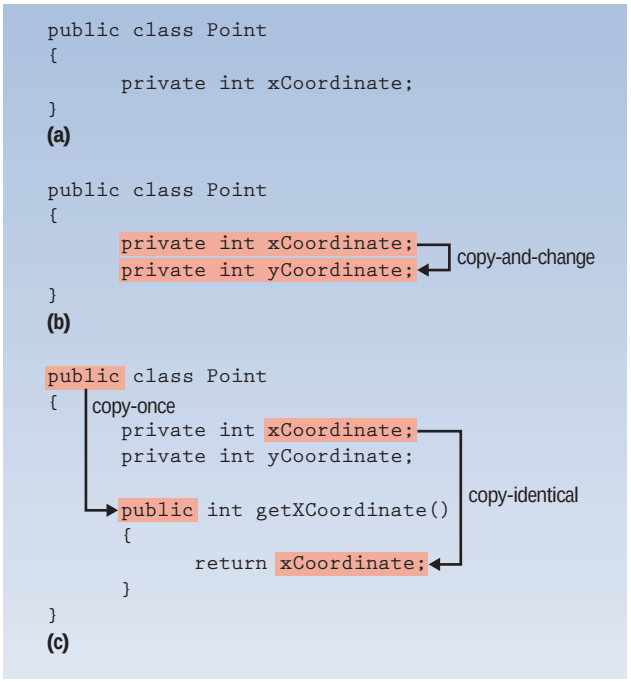

Figure 1. Copy operations used to create a Java class representing a point in the plane. (a) Original code, (b) first copy operation, and (c) additional copy operations.

An example

To illustrate how the operations might work, consider a programmer who wants to create a Java class representing a point in the plane. He might start with the code segment in Figure 1a.

At this point, the time is ripe for the first copying, as in Figure 1b. The programmer created a copy of the definition of xCoordinate and changed the name of the attribute in the replica to yCoordinate. He used the copy-and-change operation because there is a logical connection between the two lines (it is no accident that the definition of the two coordinates is similar), but of course the two lines are not meant to be identical.

Now the programmer adds a get method for one of the attributes—using additional copy operations—as in Figure 1c. Notice that he used the copy-identical operation for the return statement because the name of the returned variable must be identical to the name in the declaration of that variable. On the other hand, he copied the public modifier using copy-once because there is no logical connection between the modifier of the class and the modifier of the get method. To reduce the amount of typing, he merely exploited the coincidence that the two modifiers match.

Implementation issues

Implementing the copy-identical operation and the associated automatic updating is not trivial. The user can apply the copy-identical operation to text that contains or is contained in a block previously copied using copy-identical. Therefore the program must store two relations between the copied blocks: which blocks it must keep identical (an equivalence relation) and the is-part-of relation.

Implementing the copy-and-change operation is similar because again the text in a copy-and-change operation could contain or be part of another block copied using copy-and-change. Therefore, the block hierarchy is important in this case as well. The only difference is that the other stored relation—which block was copied from which one—is not symmetric. In the copy-identical operation, the operation's direction did not matter because the aim was only to keep those blocks identical. However, in the copy-and-change operation, the roles of the original and the replica are slightly different.

Of course, the relations that the copy-identical and copy-and-change operations define are not independent: For example, text copied using copy-identical can contain another block of text involved in a copy-and-change operation. Therefore, the program must take into account all three relations—those induced by copy-identical and copy-and-change as well as the is-part-of relation.

The program must also store these relations persistently so that it can enforce them beyond single editing sessions. Typically, this means storing them in the file that contains the document. Another possible strategy is to store the relations in an additional file to avoid polluting the actual file with this information. A source-code editor, for example, could employ this strategy, since source files must contain only program code.

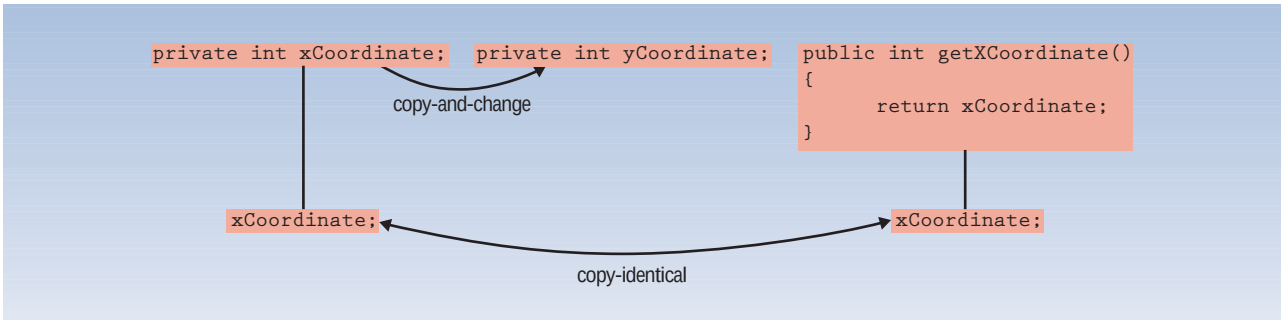

Figure 2. Internal representation of the copy operations in Figure 1. The vertical lines correspond to the is-part-of relation between the relevant blocks of text; the other edges represent the copy-identical and copy-and-change operations.

Implementing the move and copy-once operations does not pose similar challenges because the program does not have to offer any further support after moving or copying the given text, thus it does not have to record these operations.

To illustrate the persistently stored relations induced by the copy operations, Figure 2 shows the relations resulting from Figure 1. The edge representing the copy-identical operation has arrows on both ends, since copy-identical defines a symmetric relation. In contrast, the edge representing the copy-and-change operation points from the original to the replica, since copy-and-change defines an asymmetric relation. The copy-once operation does not appear in this representation because it does not affect document consistency.

Applying the relations

To illustrate the benefits of the proposed program, consider the scenario based on the previous example, in which the programmer later changes the definition private int xCoordinate to private int horizontalCoordinate. The editor program would then make two suggestions. The first would be to change the definition of yCoordinate. Because the programmer had copied the changed definition using copy-and-change to yield the definition of yCoordinate, the program would suggest that this definition might also need changing. And indeed, the programmer will probably want to change it to verticalCoordinate.

The second suggestion would be to automatically change the getXCoordinate method to return horizontalCoordinate. Because the programmer copied the block xCoordinate using copy-identical to the return statement of the getXCoordinate method, the program would suggest automatically changing that line to return horizontalCoordinate, which is exactly what the programmer would want.

Although most of the examples given here are from the programming domain, the proposed operations are also suitable for copying objects such as parts of a picture or an audio or video file or spreadsheet cells. The operations offer powerful possibilities for defining semantic relations between object copies and help ensure consistency as the document evolves. Of course, with more flexibility comes the need to more carefully consider the best option, which implies a slightly longer learning curve, but these efforts are a small price to pay for the relief of not having to worry about inconsistent text copies. This is especially true for long-lived documents.

Instead of resorting to a one-size-fits-all solution, one approach could be to use traditional copy-paste for short-lived texts and the proposed operations for long-lived texts. But using mark, move, and copy-once gives the user approximately the same functionality as traditional cut, copy, and paste. The new concepts simply scale better because of the additional copy-identical and copy-and-change operations.

All this works well and is relatively easy to implement if the user employs only one editor program to edit and copy objects within one document. All copying is in the realm of the editor program, which can thus support the long-term maintenance of the defined relations. The editor program might even provide the same support for copying across document boundaries. For that, it would probably store the interdocument relationships in a central database.

However, things rapidly become more complicated when the user must rely on multiple editor programs. Several users might be editing the same document, each with a preferred editor, or users might need different programs to copy objects between different document types, such as between a spreadsheet and a diagram. Administering semantic relations beyond program boundaries clearly requires a generic mechanism that the operating system must provide and that editor programs would access through standardized interfaces. In this respect, the situation is similar to that of the clipboard: The operating system provides a means for doing copy operations between programs. Designing such a facility for the proposed operations on the operating-system level is an important goal for future research.

References

1. B.S. Baker, "A Program for Identifying Duplicated Code," *Computing Science and Statistics*, vol. 24, 1992, pp. 49-57.
2. S. Ducasse, M. Rieger, and S. Demeyer, "A Language-Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance*, IEEE Press, 1999, pp. 109-118.
3. T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large-Scale Source Code," *IEEE Trans. Software Eng.*, vol. 28, no. 6, 2002, pp. 654-670.
4. J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance*, IEEE Press, 1996, pp. 244-253.
5. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
6. M. Kim et al., "An Ethnographic Study of Copy and Paste Programming Practices in Object-Oriented Programming Languages," *Proc. Int'l Symp. Empirical Software Engineering*, IEEE CS Press, 2004, pp. 83-92.

Zoltán Ádám Mann is a doctoral student in computer science at Budapest University of Technology and Economics. His research interests include software engineering and embedded system design. Mann received an MS in computer engineering from Budapest University of Technology and Economics. He is a member of the IEEE. Contact him at zoltan.mann@cs.bme.hu.