15

Cut, Copy, Paste, Drag and Drop

When UNIX introduced the concept of a pipe, which carried information out of one program directly into another program, it fundamentally changed the way people thought about programs. It allowed problems to be solved by piecing together the capabilities of various pieces of code rather than rebuilding from scratch. A similar transition occurred when the Apple Macintosh introduced Cut, Copy and Paste to the world at large. It suddenly became trivial for users to combine the work and interactive capabilities of a number of unrelated programs to create unique solutions.

All of these mechanisms involve the transfer of data from a *source* to a *destination*. In many cases the source and destination share the same data model in the same program. Examples of this would be cutting a paragraph from one part of a document and pasting it into another part, or making many copies of a shape to distribute around a drawing. Throughout most of this discussion we will assume that the data transfer is between two different programs because that is the most general case. There are three issues that we must address. 1) What is the appropriate data transfer to perform? 2) What is the transfer mechanism? 3) What should be done with the transferred data? We will address each of these in turn.

Appropriate data transfer

When the user requests a transfer between source and destination, we must determine the kinds of data the source can generate, the kinds of data the destination can receive and the intent of the user. This requires some communication among programs about data types and possibly some additional input from the user. For purposes of discussion we will use the simple clipboard model of data transfer where a source puts information onto the clipboard (cut or copy) and a destination retrieves information from the clipboard (paste).

A problem with such transfers arises when the source and destination do not have the same data representation. For example, copying from WordPerfect and

pasting into Microsoft PowerPoint can cause a problem. The source and destination must establish a common representation that both can understand.

The simplest mechanism is for the source to generate all of the different forms of data that it can. So when the user requests a Copy of a paragraph from WordPerfect, it may place the information on the clipboard in WordPerfect format, plain text, HTML and possibly Microsoft Word. By placing all forms that WordPerfect understands onto the clipboard, any destination programs can then choose for themselves.

The Macintosh style guide and all subsequent style guides have mandated that every source must produce either plain text or an image as one of its formats. Every destination should accept both of these representations. The reason for these requirements is that it virtually guarantees that copy and paste will always work in a form visible to the user. It is very important for usability that users believe that cut/copy/paste will work virtually anywhere. Any information in a graphical user interface can be captured either as text or an image.

Limiting ourselves to just text and images would not be very satisfying. When we have formatted some text or a drawing we would like that formatting to be retained. When operating in specialized applications, we want all of the information structure to be transferred rather than just a picture.

Types

When placing information on the clipboard the source specifies the *type* of each of the clips. In the days before object-oriented programming this type was a string such as “TEXT”. In the very early days the type name was limited to four characters so that it would fit in a word of memory. To perform a Cut or a Copy a program opens the clipboard and clears out all existing scrap. It then puts as many different types of data in as it wants. With each type of data there can be a type name and the bytes of content information. By publishing multiple copies of the data in different formats on the clipboard the source has made its capabilities available to the destination. By convention the source should place types of information on the clipboard in most preferred to least preferred order.

To perform a Paste, the destination consults the clipboard for the types of information that it can understand. By looking at the order of the information on the clipboard, the destination program knows the representations preferred by the source. The text/image requirement ensures that the destination program will usually find something useful.

It is universal for cut/copy/paste that the user interface toolkit will provide access to the clipboard and a mechanism for a source to place many clips of data on the clipboard in various types or data formats. The names for these source mechanisms vary widely but the all of the same pattern of:

- access the clipboard
- clear out existing items
- repeatedly add type/content pairs to the clipboard
- close the clipboard and make it available for use.

Toolkits also provide the destination with mechanisms that will

- access the clipboard
- get a list of the types or data formats currently available on the clipboard in the same order as they were added by the source.
- get a data object using a particular type or data format.

The transport mechanisms and the ways of naming and processing types may vary but these basic strategies are almost universal.

Naming Types

Naming of types is a bit of a challenge because the source and destination must have the same understanding of type names. Apple provided a central registry of type names so that various companies would not accidentally reuse the same names.

Outside of personal computers the Internet has similar problems with transfer of information via email and web pages. In an Internet-oriented computing world the MIME (Multipurpose Internet Mail Extensions) types are commonly used. These are textual names for various kinds of information. MIME type names generally consist of a content type, such as text, image or video, and a subtype such as jpeg, gif or tiff. As with the simple clipboard type names used by the Macintosh, there is a need to register MIME type names so that their meaning is universally understood. This is handled by the Internet Assigned Numbers Authority (IANA)[1].

With the advent of object-oriented programming it is frequently desired to pass complex objects between various parts of an application or between cooperating applications. Object-oriented languages that support reflection, such

as C# or Java, allow clipboard entries to use the Type or Class of the object as a type identifier. In such languages it is possible just to put an object onto the clipboard and let the system determine its type. This mechanism allows objects to retain their representations. In Java, for example, the type is sufficient to process the object so long as the other application has that class on its CLASSPATH. In C# similar things are possible using DLLs and/or assemblies of code. This mechanism is very convenient for integrating programs because programmers can now think exclusively in terms of model objects. However, the Type or Class mechanisms are confined to transfer among programs using the same language. This is convenient but very restrictive. The more general case is the textual type names and in particular the MIME types.

User intervention

In some cases the transfer intent is not clear from the source and destination types. Suppose one copies a section of a spreadsheet. Later when pasting that material into another part of the spreadsheet or into a different spreadsheet there are some ambiguities. Are just the visible values of the cells to be pasted or are the formulas for computing the values also to be pasted? When pasting formulas that reference beyond the copied region, what is to be done with those references? Should the format of the source cells be used or the formatting of the destination? These kinds of questions cannot be answered exclusively from type information because all of the options are acceptable to the software. The question is the intent of the user. The user interface convention is for Paste to use a default that preserves the most information and then provide a Paste Special option that allows the user to choose from the available possibilities.

Making the connection

Conceptually the clipboard is a shared memory space. The source places items on the clipboard and the destination rummages through them to find useful information to copy from the clipboard. However, there are several issues that complicate this model. First, there are differences depending on whether an object-oriented programming language is assumed or a lower level language such as C. Second, there is the question of what boundaries exist between source and destination. These include crossing process boundaries, programming language boundaries and hardware system boundaries. Third, on smaller machines, space for all of the various formats of the same item can be an issue. This is less important than it once was due to larger memories. Fourth, there is the problem of the time to render an item into all of the various formats that might be useful.

If a source renders a copied item into 10 formats, it is rare that more than one will be used in a destination.

The simplest clipboard mechanism is to serialize the copied or cut objects into all of the possible forms. This makes the clipboard into a collection of type-named sequences of bytes. This requires the source to generate byte streams in all of the various formats that might be desired. The first improvement on this primitive model was *deferred posting* to the clipboard. In X-Windows this was accomplished by an application posting itself as the clip source and then responding to any events requesting clipboard information. Clip contents were not rendered until a particular format was requested. Microsoft Windows allowed source programs to post a type with a null pointer to the data. If the destination requests a type for which the pointer is null, the system sends an event to the source program instructing it to render the desired type into the clipboard where it is then available to the destination.

The advantage of deferred posting is that only formats that are actually used need be generated. The disadvantage is that the source must remember what is on the clipboard for as long as it owns the clipboard information. In many cases the source program has moved on to perform other edits or operations. However, for the clipboard system to work it must remember the information. One of the nice features of the Microsoft Windows solution is that the source has either option. It can render all of the types and forget about the clipboard or it can defer and remember what it needs. A compromise strategy is also possible. The source can render clip information in its native form, post null pointers for all other types and then if necessary render alternative forms from the native information posted on the clipboard. This has the advantages of deferred posting without the problems of remembering clip information that no longer exists in the source model.

The above approaches assume that the clip information must be serialized. Serialization means that model objects are rendered into a self-contained stream of bytes. This is required if the source and destination are two different processes with different memory spaces or even more so in the X-Windows case where the source and destination may not even be on the same machine. However, a common use of cut/copy/paste is from one place in an application to another place in the same application. It would be helpful just to move the data objects without the serialization step.

Object-oriented cut/copy paste

Object oriented languages can simplify many of the issues in cut/copy/paste. The first step is to wrap the source of clipboard information in an interface. In C# this is the IDataObject interface. In Java/Swing there is the Transferable interface. These interfaces provide methods to find out the types of information available and to get information of a particular type. Any source that conforms to the interface is appropriate. By implementing IDataObject or Transferable a source has complete freedom to decide how the clipboard information will be saved and rendered. The methods subsume any event handling mechanism. The destination need not be concerned with any of these issues. It is presented with an object that conforms to the interface and uses it to get the information that is needed.

A source can create an IDataObject that simply stores the data in process memory. If the destination is in the same process space, then pasting is just a copying of objects or pointers to objects. If the destination is in a different process then the clipboard system presents the destination with an IDataObject that will work through shared memory and the event mechanism to transfer data across process boundaries without either the source or destination needing to know. If the source and destination are on different machines, IDataObject or Transferable proxy objects can handle the communication without either side being involved. This approach allows the source, the destination and the clipboard system to all perform their functions relative to a simple interface without much effort on the part of any of them.

A second advantage of object-oriented languages with reflection is the use of language types as the clipboard types. If the source and destination both share the same language and class definitions then the type naming can simply use the class objects rather than MIME types. This greatly simplifies transferring all kinds of complex objects in a very natural way. The MIME types or similar naming will still be required when information must be transferred to applications written in different languages or possessing different class definitions.

Lastly reflection-based languages can provide default serialization of objects. Rather than the source application translating its objects into a stream of bytes and the destination parsing those bytes, many languages will automatically perform this function using the class object information. In reflection-based languages, every class is known at run-time and every field in every class is known also. This allows system developers to write a single method that will explore the class information of any object and render a serialization of that object and any other object that it references. Similarly a general parse method

can be written to reconstruct an object from the stream. This allows source and destination applications to completely ignore everything except posting and getting objects, with the language system taking care of everything else.

There is, however, a warning about automatic serialization. The default automatic mechanisms work by recursively following every pointer to find every part of a complex object. This is particularly a problem in user interfaces. If every model object has listeners to be notified when the model changes, then those listeners are pointers to objects. The automatic serialization will follow those listener pointers to the widgets displaying the model. We rarely want the widgets serialized with the model but it gets worse. Every widget has a pointer to its parent widget. Every parent widget has a pointer to all of its children. Every child widget has a pointer to its model. By implication serializing a simple model object has the potential of serializing all application data and the entire user interface. This is not a good plan. Most serialization systems have mechanisms for controlling what pointers are followed and what actually gets serialized. With a little attention this problem can be addressed.

Source/Destination relationship

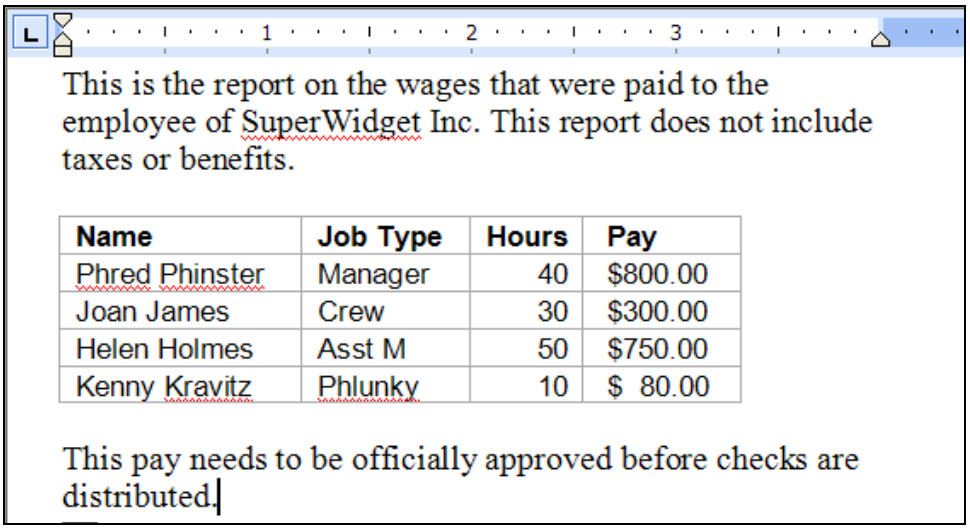
When an object is copied or cut from a source and placed in a destination there are three forms of relationship that may be established. The first is that described above where the data is copied and the relationship to the source is forgotten. The pasted data is now the responsibility of the destination. The second form is where the data is copied in its native form and a link is retained to the source so that the source application can be used to change the data. The third is where a link to the data is what is copied as well as an image of the data. Whenever the source is changed, the pasted data in the destination is also changed. The first form has already been discussed.

Book1

	A	B	C	D	E	F	G
1	**Name**	**Job Type**	**Hours**	**Pay**		**Job Type**	**Wages**
2	Phred Phinster	Manager	40	$800.00		Asst M	$15
3	Joan James	Crew	30	$300.00		Crew	$10
4	Helen Holmes	Asst M	50	$750.00		Manager	$20
5	Kenny Kravitz	Phlunky	10	$ 80.00		Phlunky	$8

Figure 1 – Spreadsheet to copy and paste

The second two techniques can be illustrated by copying a portion of a spreadsheet into a word processing document. In the Microsoft world this is called Object Linking and Embedding (OLE)[2]. Suppose we have a spreadsheet like that shown in figure 1. We want to copy the shaded region and paste it into a document, as shown in figure 2.



This is the report on the wages that were paid to the employee of SuperWidget Inc. This report does not include taxes or benefits.

Name	Job Type	Hours	Pay
Phred Phinster	Manager	40	$800.00
Joan James	Crew	30	$300.00
Helen Holmes	Asst M	50	$750.00
Kenny Kravitz	Phlunky	10	$ 80.00

This pay needs to be officially approved before checks are distributed.

Figure 2 – Spreadsheet data pasted into a document

At this point the data has been pasted into the document and displayed. Pasting a picture of the data would be sufficient for this purpose. However, data tends to change. Suppose that the manager only works 35 hours instated of 40. We could go back to the original spreadsheet, make the change and then repeat the copy/paste process. However, this would require that the owner of the document could find the spreadsheet.

An alternative is to store the original native spreadsheet data in the document along with the name of the program that manages that data and a picture of the data. For most document uses the picture is shown. The destination application has no knowledge of how the data is formatted, it just displays the image. When the user double-clicks on the table in the document and it changes as shown in figure 3. Knowing the application that understands this data the host has created a window where the data was originally in the document and has launched the source application passing it the data and the window. This allows the source application to edit the data using its own user interface. When the editing is done, the window is deselected, the new data and image replaces the old and the

document returns to normal. This *edit in place* strategy allows a destination, such as a document, to play host to a wide variety of editable information without knowing anything about how those applications function other than the name of the application.

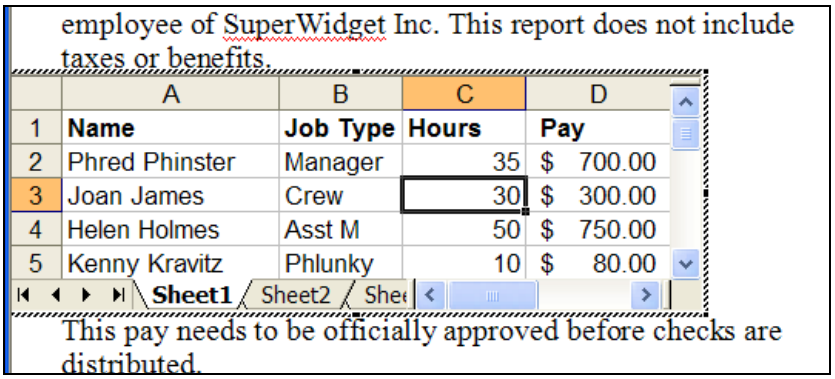


Figure 3 – Edit-in-place

When doing pasting in this fashion there is a question of what should be pasted. In figure 4 we show that the table that defines the wages is included in the pasted data even though it does not show in figure 2. Considering the way spreadsheets work this is important because pay cannot be computed from hours without the wage information. When copying from the spreadsheet the source must consider the information needed for later editing.

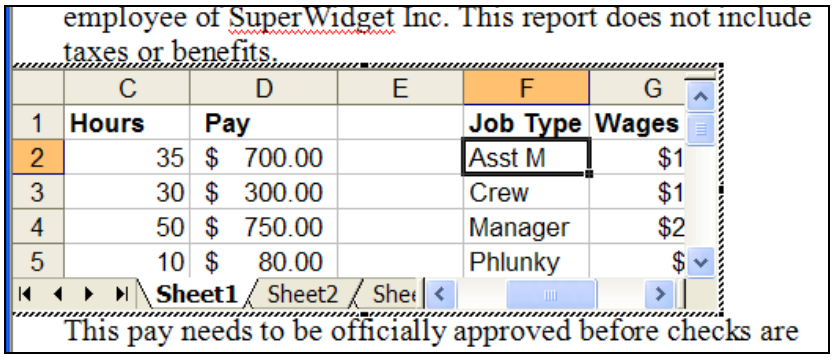


Figure 4 – Scrolling the pastes spreadsheet

There is an alternative to edit in place called *edit aside*. In edit aside the host application opens a new window of its own with its own menus and toolbars. For a source application to support edit in place it must prepare a special view that

works well in the embedded situation. Frequently this involves mixing the source program's menu items with the destination program's menu items to provide the necessary functionality to the interactive user. The edit aside option is much simpler for the source application. The user interface is unchanged. The only thing that is different is saving and loading information from within the destination's paste location rather than a file.

The third source/destination relationship is a link to the original source data. Suppose that our spreadsheet from figure 1 was part of a much larger wages report for all subsidiary companies. We would want our simple document to change whenever the larger report changes. This would involve pasting only an image and a link to the source in the destination. Whenever the source is changed, the destination would follow the link get the source to generate a new image and paste that image in. This was pioneered by Apple's Publish and Subscribe facility. The problem with such links is that, like web-page links, they get broken. They also have a problem with robustness. Suppose that the pasted table in figure 2 was implemented as a link and the source application inserted a new column. If the link was defined as C1:F5 and a new column was added, the pasted material is now linked inappropriately. It is possible that the source could remember all published links and update them, but that complicates the source program.

Layout of embedded information

When pasting embedded information there is a problem of layout. Suppose that the table in figure 2 were to cross a page boundary. If this were a normal word processor table an appropriate break could be found at one of the table rows. However, if the pasted table is data from another application such page break behavior is much more complicated because it requires some negotiation between the destination and source application. Apple's OpenDoc[3] system attempts to address this problem. Microsoft's OLE, however, assumes that embedded material occupies a rigid rectangular window and does not attempt to accommodate more sophisticated relationships. Because OLE is simpler to create and simpler to use it made it to market much sooner and was more widely adopted.

Drag and Drop

A very popular variant on cut/copy/paste is drag and drop. Drag and drop moves data from a source to a destination but it does not use the conceptual intermediary of a clipboard. Underneath many of the data transfer mechanisms

are the same, but to the user there are some differences. There are three issues from the user interface perspective. They are: 1) how do we determine in the user interface when drag and drop has been initiated, 2) how do we provide feedback to the user during dragging and 3) what does drag and drop actually mean?

Consider the drawing program shown in figure 5. The mouse is over the image and on mouse-down/mouse-move it will begin a dragging operation. The question is whether this is a simple translation or is it the start of dragging this image into another picture. In the case of the picture we would probably decide that this is a translation until the user starts to drag outside of the picture boundaries in which case we would switch to a drag/drop operation. In most systems it is up to the source application to decide when a drag/drop operation should begin. When a source does decide, it can offer its data for dragging in much the same way as with putting information on the clipboard. In Java/Swing this is done using the same Transferable class to encapsulate transferable data.

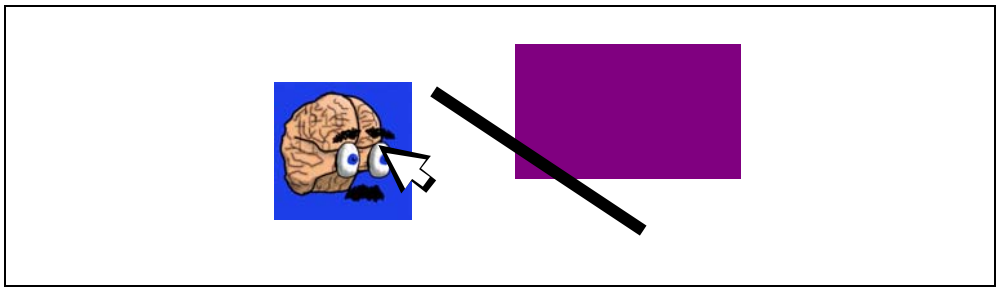


Figure 5 – Drag-and-drop image

There are three entities that participate interactively in a drag and drop operation. The source widget initiates the interaction and must provide feedback to the user about the object being dragged. Potential destination widgets must provide feedback about whether they will accept a drop of a particular data objects. Lastly the drag/drop system must provide feedback about the object being dragged.

The interactive response of the source is generally to show the source component as highlighted. This is relatively straightforward. To handle the feedback by the destination the Widget interface generally provides events that notify Widgets when there is a drag operation in progress over that widget. In C# there are the following:

- DragEnter – an object being dragged has just entered the widget.
- DragOver – similar to a mouse-move only with drop data attached.
- DragLeave – the dragging object has left this widget.
- DragDrop – the user has released the mouse over this widget to initiate a drop.

In addition most systems have a flag on a widget to indicate whether that widget will accept drops at all. This allows the drag/drop system to ignore widgets that are not interested. Each of these events includes parameters with information about the mouse location, the source of the data, and the Transferable data container itself.

There are two common scenarios for the destination. The first is to examine the data on DragEnter to see if any of the information is acceptable. This works much like examining the clipboard in a paste operation. The destination widget then responds with user feedback on the kind of drop operation it will perform if dropped. In C# a widget can choose standard feedback options from among Move, Link, Copy or None. The decision among these is generally based on the data and the mouse button that was pressed. With this simple strategy, the drag/drop system handles all of the feedback, usually by changing the cursor to indicate the operation.

The second approach is that the destination will modify itself in some way to reflect the consequences of the drop if it is performed. Figure 6 shows the Eclipse IDE dragging a view around the screen. The mouse cursor at the right has been changed to show where the new view will be placed and the shaded rectangle shows how the existing window space will be split to make room for the new view if it is dropped at that location.

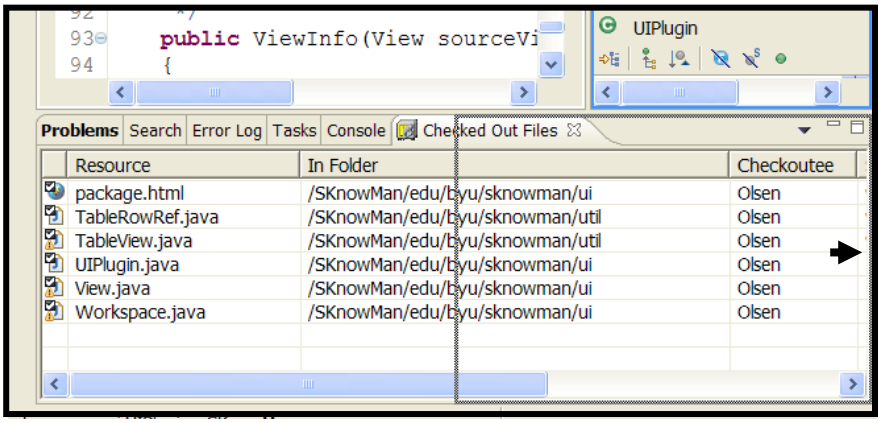


Figure 6 – Dragging views around Eclipse

With Cut/Copy/Paste it is usually at the source where the ultimate action is decided. For example a Cut will remove the data from the source and place it on the clipboard, while a Copy will make a duplicate. With drag/drop it is generally at the destination where the action is decided. This is primarily because in some cases the user expresses an action by the location where the drop is performed. Dropping into a trash can implies a deletion from the source rather than just a copy. In some cases the destination will pop up a menu after the drop to allow the user to specify the actual action to be performed.

Summary

Cut/Copy/Paste and Drag/Drop provide mechanisms for the user to transfer information among applications and across networks. There is always an intermediary such as a clipboard or Transferable object that serves as the container and negotiator between source and destination. The source indicates the data formats that it can deliver and the destination takes what it wants.

---

[1] http://www.iana.org/

[2] Brockschmidt, Kraig Inside Ole, Microsoft Press, (May 1995)

[3] Apple Computer Inc. OpenDoc Cookbook, Addison Wesley (February 1996).