The Case for an Open Data Model

Brad A. Myers

August, 1998
CMU-CS-98-153
CMU-HCII- 98-101

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Abstract

The trend in modern software systems such as Java is to support "reflection" where independent software can query to find out the properties of objects. We have been investigating the implications of taking this property even further—so that all aspects of an application are open and available to inspection by external software. By making the fundamental data structures of the application have a standard format, external components can access the information they need without requiring a complex protocol. We have found that this gives the application developer and end users many important benefits, including support for increased automation, extensive end-user customization capabilities, external agents and tutors, sophisticated search and replace, scripting and macros, alternative interfaces without re-implementing the application, plug-ins that operate in the same space, and significantly higher re-use of common code. Many of these benefits are demonstrated in our Amulet user interface development environment which uses the open data model.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

Keywords: Open Data Model, Object-Oriented Programming, Components, Toolkits, User Interface Tools, User Interface Development Environments, User Interface Management Systems (UIMSs).

1. Introduction

One of the reasons that the World-Wide Web spread so swiftly was the early adoption of a common, standard data representation: HTML hypertext files and the GIF picture format, which all Web applications can use and generate. This relatively simple format allows people to use browsers from multiple vendors, enables many different ways for people and programs to generate pages, and it allows computer programs such as search engines to search and analyze pages fairly easily. The pipe mechanism in the Unix operating system is successful for a similar reason: a common data format. In Unix, applications generate and accept plain ASCII text, so one application can easily process the output from another.

This is quite different from the world of today's desktop applications, however. Each application, such as the text programs MacWrite, Microsoft Word, Pagemaker, and Framemaker, and the drawing programs MacDraw, Adobe Illustrator, and Microsoft PowerPoint, use their own proprietary data structures, both on disk and in memory while the application is running. This allows the data structures to be optimized for each application, but it prevents a number of useful benefits that would arise if the applications used a common data format.

This article argues that if all applications used a common data format, enormous benefits would result, both for the application developer and for the end user. First, the characteristics of the "open data model" are presented, followed by a discussion of the benefits. The article then discusses how various current systems support open data.

We have been investigating the implications and implementation issues of using the open data model for the last 10 years in our Garnet [18] and Amulet [20] user interface development environments. Our current results are discussed next. Finally, some of the potential concerns with the open data model are also discussed.

2. What is an Open Data Model?

In a completely open data model, the important application data structures would be in a standard format that is completely accessible to all programs. The data could then be read and manipulated by external programs that are not necessarily developed by the same vendor.

Note that this goes beyond just having a standard format for disk files. External applications need to access and manipulate an application's data structures *while the application is running*. The external application also needs to know what *operations* are provided for operating on the data and to have the ability to monitor and execute those operations. This includes every operation that is available to the user, as well as additional monitoring and manipulation actions. As an example, when an open data model is used for a drawing program like MacDraw or Microsoft PowerPoint, then an external application could find out what objects are currently in the window and what their properties are (“rectangle_001: position=(10,30,50,80) and color=red, circle_002: position=(100,100,200,200) and color=blue”, etc.). Also available would be the full set of operations available so the external application could manipulate the objects in any of the ways available to the user (“delete rectangle_001”). The application would also be able to monitor the user’s actions, and be notified whenever the user performs any operation. In the object oriented community, the ability to find out the properties of objects at run-time is called “reflection.” Reflection allows the class of any object to be discovered, along with all of the public methods and data. Microsoft’s OLE provides related facilities through the “OLE Automation” interfaces.

While reflection is important, it is not sufficient. The system must also have a set of standards about what data will be called and what types will be used. For instance, if a graphical object calls its position “location” or even worse, “ubicación” instead of “position,” then other software will not be able to discover where the object is on the screen even if the names and values of all slots are available.

Also required is the ability to investigate the *behavior* of an application. Although reflection allows you to find out the methods of an object, this is not sufficient. We want to inquire about what operations the *user* can do and the results of those operations. Therefore, standards are also necessary for how operations and their parameters and results are represented. For example, it must be possible to answer questions such as: what are the commands in all of the menus, what does clicking the mouse in a window do, how are objects moved, etc. Without a standard for representing this information, there is no way to know which of the hundreds (or thousands) of methods and object types are relevant to answering the question.

Lieberman’s defined the terms *scriptability* and *examinability* [12], and the open data model supports both. “Scriptability” is available because external applications can record the user’s actions and execute any recorded command. Therefore a scripting mechanism can easily record macros or scripts and execute them later. “Examinability” is available because external programs can examine and modify the data structures of an application. Examinability implies a level of standardization of names, as discussed above, and therefore goes beyond just reflection.

One way to provide an open data model is to define a standard data structure, require all programs to use it for their internal data, and export the data structure. For example, all information could be encoded using "attribute-value" pairs, where the names of the attributes describe the purpose of the data, and the value itself is tagged with a type inspectable at run-time. For example, in our Amulet toolkit (explained in section 5), all objects are defined by attribute-value pairs, and a rectangle in a drawing program might be defined in part by:

Slot Name	Slot Value	Slot Type
Prototype:	Rectangle	Object
Name:	"rectangle_001"	String
Left:	10	int
Top:	30	int
Width:	50	int
Height:	80	int
Color:	red	Style

Amulet defines a set of standard attribute names that are used for specific properties whenever they appear, and applications are free to define new attributes as needed. Routines are available for querying an object to get the list of the attributes, the values of the attributes, and the types and data for the values. Behaviors are inspectable in Amulet because special "Interactor" and "command" objects are used to represent all actions.

For the open data model to provide the maximum value, all applications will have to use it. Therefore, the facilities that support the open data model should be provided by an underlying toolkit or operating system that is used by all applications.

3. Benefits of an Open Data Model

There are many benefits that result from a widespread use of the open data model. Some of these have been demonstrated in existing systems, and others are more speculative.

3.1. Automation

Michael Dertouzos argues in his new book *What Will Be* [3] that to enable communication and automation, all applications should be able to provide and accept attribute-value pairs of information, which he calls "e-forms." Rather than requiring users to go to many different computer programs and web pages to enter and read information, these programs could communicate with each other using e-forms. He calls for the standardization of certain field names, with other fields specific to certain industries or groups of applications. Although his complete scenario requires sophisticated intelligent agents and speech recognition software, the e-forms are the glue that allow these to work together. He claims that the result will be a productivity gain of 200 to 1 [3, p. 86].

3.2. End-User Customization and Macros

The open data model will help support end-user customization. Some applications, like Microsoft Word, come with a sophisticated customization sub-system that allows users to change the menus and keyboard accelerators for commands. However, most applications still do not have such a facility because it is difficult to build. If the application were written using an open data model, however, then a general-purpose customization subsystem would be able to find out all the commands available to the application, and would be able to query and modify the menus, toolbars, and keyboard accelerator lists. Therefore, the customization facility could be available in the underlying toolkit, used by all programs. Users would have this ability in all applications, and it would operate in the same way for each one. Furthermore, programmers would not need to re-implement customization dialogs for each new program.

The next step is to allow end users to *create* their own commands. Many applications allow the user to create a "macro" or "script" by going into record mode, executing some operations, and then stopping the recording. Some Microsoft products (e.g., Word and Excel) support macros by constructing a Visual Basic program from the transcript. The open data model can allow the scripting mechanism to be much more independent of the application, since the scripting mechanism can monitor the commands that are executed by the user, and can query the commands themselves to discover the appropriate operators and parameters to record in the script. For example, Amulet's scripting facility is built into the toolkit and therefore is available to all applications without requiring any extra work from the programmer [16].

More than just recording the exact transcript of operations, the recording system could also be "intelligent" and look for repeated operations. For example, the Eager system [2] continually watches the user's actions in HyperCard and pops up an icon to propose that the system finish the job when it detects a repetitive action.

Using an open data model means that there is a programmatic interface to the objects and operations of all applications, so it becomes easier to supply a programming language directly to the users, in the way that Visual Basic is available for Word and Excel. Third parties could use the programming language interface to make highly customized versions of applications for sub-markets, in the same way that there are now specialized CAD programs for many kinds of design which have been created using AutoCAD's AutoLisp and C extension languages.

3.3. Agents and Tutors

When the operations of an application are observable and executable by other programs, as they need to be to support the macro and customization facilities discussed above, then it becomes much easier to supply *tutors* and *agents* for applications. An intelligent tutorial could guide the user through an application, presenting information about each operation, and monitoring whether the user performs the expected operations. If the user performs a different operation, then the tutor could inspect the objects in the application to see if the result is close enough. Because the tutor could discover *where* objects are in the window, it is easy to support highlighting as in AppleGuide to direct the user's attention, even to things that change position. Intelligent "Tip Wizards" could be created that would look for inefficient or inappropriate operations and pop up a helpful dialog window (this could be linked to an Eager-like system that would offer to perform the predicted operations for the user, as discussed in the previous section). Whereas creating such tutors and agents might be a big job, they would only have to be created once and then could be used for all applications, since the open data model means that all of these facilities could be implemented without affecting the structure or coding of the application itself.

Some of these capabilities are already demonstrated by the Microsoft Office Assistant help agent (the "dancing paper clip" in Office'97). The Office Assistant needs to monitor user's actions and access the application's current state. The "Show me" option available in some help messages performs the operations using the application's actual menus and dialog boxes, so these must be interpretable to the program, even if the user has customized the menus and moved menu items around. The Office Assistant has deep knowledge about the implementation of the office applications. Using an open data model would make this kind of agent available for all applications, and would make it easier to connect it to the application.

Another issue is the agent's ability to *change* the user interface of an application. An open data model allows the agents to be external to the application and still change the application's user interface since the widgets[1] are inspectable and manipulable externally. As an example of why this might be useful, Eager [2] adds an icon of a cat that can be clicked on, and it changes the color of one of the application's menu items to show what the agent thinks the user will do next.

---

[1] A "widget," sometimes called a "control," is a way to input a particular type of value. Examples of popular widgets include menus, scroll bars, buttons, text input fields, toolbars, etc.

3.4. Rich searching

One of the most useful of the Unix utilities is “grep” which searches through files to find the ones that contains a particular string. The reason this works is that most files in Unix are plain ASCII text, so a general-purpose search mechanism can look inside any file. However, PC and Macintosh applications use a variety of file formats, and very little information on a typical PC or Macintosh is stored in ASCII files. Earlier windowing systems therefore did not let you look for text inside files, but Windows 95 now supports this by using heuristics to find the ASCII content of files. Since it uses heuristics, it can miss finding some matches if the application does not store the text contiguously. For example, the RTF and Postscript disk formats break up words if a letter has different formatting from the rest of the word. A better approach is to use the open data model for files, so that all applications will use a standardized model which can be understood by any application, including a smart search engine. This would also enable searches on more than just ASCII text, such as “Find all the presentations that use this picture,” or “Find the pages that do not have the word ‘CMU’ in red in the header.” Advances in image processing and natural language understanding would immediately be available for searching across all documents from all applications, because the search engine would be able to identify the appropriate parts of the documents to process.

Whereas every text editor has find and replace commands, very few graphical programs can search for graphical objects. Kurlander showed how to create a general search and replace mechanism for graphics in 1988 [11], but this has never been provided in a commercial program. One reason is that it is fairly complex and would have to be reimplemented for each application in a conventional architecture. With the open data model however, such a search and replace tool can be provided in the library and used by all applications since it could look for and replace any property. For example, Amulet’s graphical search tool works with any application because it can automatically determine and query the properties of all objects (see **Figure 1**).

3.5. Beyond spell-checkers

For a spelling-checker to work, it must go into the internal data structures of the text editor and find the words. The open data model allows this facility to be supplied easily, and also to be significantly extended. Of course, spell checkers could be written once and used in all applications, even in graphical applications (“check the spelling of the comments of all my circuit elements”). In addition, other consistency checkers could be developed. Some ideas are that a color checker might check whether all diagrams used a standard color scheme, and a size checker might check whether all presentations will be viewable when projected on a big screen at a conference.

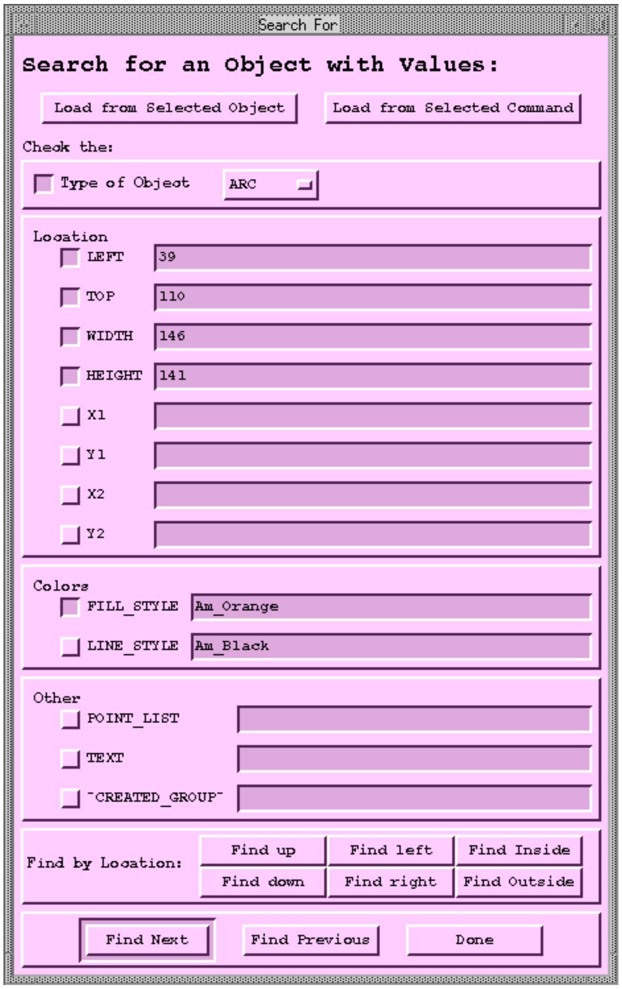

Figure 1. The search (find) dialog box automatically created by Amulet for a simple graphical editor. All the properties for all of the types of objects are shown. The values from an arc (oval) have been loaded.

3.6. Semantic markings

In computer programs such as Microsoft's NetMeeting for use by multiple people, some of whom are on remote computers, there are usually *telepointers* to show where each user's mouse is located. However, some applications (unlike NetMeeting) allow each user's screen to be organized in a *different* way. For example, World-Wide Web browsers put words on different lines depending on the width of the window. If one user is pointing at a word in a browser window on one machine, his or her telepointer will be in the wrong place when viewed on a different machine in a window with a different width. A solution is to have the telepointer know about the content so a semantic mapping can be performed on the telepointer to make it always point to the appropriate word on each computer [5] (see **Figure 2**). A related problem is when remote user or program wants to highlight or select an object to bring it to the attention of a user [23]. With conventional architectures, the telepointer or highlight must be reimplemented for each application, but with an open data model, telepointers and highlights can be implemented separately from the applications since they can interpret the data.

a) Saul's view b) Mark's view

Figure 2. In a collaborative Web browser for use by multiple people, each person's telepointer must be placed in an appropriate place on everyone else's screen. This requires knowing where the individual words are since browsers reformat the text based on the current window width. (Figure reproduced from [5] with permission.)

3.7. Alternative interfaces

One of the big problems reported by Mynatt [22] when trying to make a graphical user interface accessible to blind people is that it is difficult to query about what operations are available to the users. The goal was to provide a screen reader which mapped properties of the widgets, such as the label, location, and whether it is grayed out or not, to properties of the sound. Existing toolkits are not designed to allow external programs to ask which widgets are available, to investigate the properties of those widgets, and to execute the widgets as if the user had pressed the mouse. Furthermore, although in Motif, Windows and the Macintosh, external programs can get some information about the widgets, the parts of the interface that are *not* widgets are completely unavailable externally. For example, in a drawing program like MacDraw, the screen reader would have no access to the contents of the main windows, because they are drawn directly by the program without using built-in widgets. If the widgets and the application's custom output used an open data model, then *all* aspects would be available to the screen reader, and the alternative interfaces would have full access and control.

Similar problems arise when trying to add other kinds of new interfaces. A number of speech systems are emerging that allow users to talk to their existing applications. For example, DragonDictate [4] and IBM's ViaVoice [9]. Because the system menubar is easy to access, these systems allow the user to say the names of the menu items, rather than requiring the use of the mouse or keyboard accelerators. However, other operations cannot be executed using speech because the speech system cannot detect what commands are available and what the user can do. An open data model would allow the speech system to let the user give *all* the commands in the interface. Most sophisticated research systems with speech or natural language interfaces have to be completely implemented with these technologies in mind, since users often want to discuss objects in the interface by their properties, for example, by saying "Delete all the red rectangles." If the objects in the system are described with an open data model, even systems created with only a conventional direct-manipulation interface could have sophisticated operations supported by add-on speech and natural language interfaces, since the speech and natural language sub-system would be able to search to find the objects which match the phrase, and then would be able to execute the specified operation.

3.8. Ability to have significant plug-ins that work in the same space

In the conventional component model, as supported by OLE, OpenDoc and Java Beans, each component controls its own area of the screen. The main goal is to allow a component to be embedded inside another component. There is another model of composition, where all operations operate on the *same* part of the screen. This is available to a small extent for third-party "plug-ins" for specific applications like Adobe PhotoShop, and has been promoted by Jef Raskin as an overall system architecture [26]. For example, if you load in a spreadsheet capability, this can operate on numbers anywhere in a document, not just in a rectangle that is called the spreadsheet part. The spreadsheet capabilities could be used to compute the labels or even the positions of objects in a drawing. As another example, if you have a simple drawing editor like MacDraw, and you need a new kind of spline-curve, instead of throwing away your editor and buying a new one like Adobe Illustrator, as would be required in all of today's architectures, you could instead just buy a spline capability and add it in to your existing graphics editor. In this design, each command in a menu and each capability of an application might be its own separate "micro-component," all able to work together on data created by all other micro-components. This might help usher in the age when small pieces of functionality can be created and purchased separately and combined by end users.

The open data model makes this vision possible, since all of the components can query and modify the common data. New components can be created by different vendors and can be dynamically added in, and they can all share and operate on the same document. They can even share the same menubar, rather than requiring the menubar to switch contents as the user clicks on different components as in today's models like OLE. In the open data model, a common operation like "Cut" can query the various selected objects and execute the appropriate operation for each type of object. In this way, the fact that there are different "components" could be entirely invisible to the user.

An analogy to the Unix pipe mechanism is relevant here. In Unix, there are many small utilities which users string together to accomplish their own custom tasks. These work because most files in Unix are plain ASCII text and the small utilities can read and emit text. For example, "wc" is a general word (and line) counting program that can be used to count the number of files in a directory, the number of words in any document, etc. In Windows, Microsoft Word needs a custom word counting tool, since no utility could work in Word and other word processors. Framemaker for Windows does not have a word counting tool.[2] With the open data model, these small utilities could again be written since they would operate on the appropriate format of data from any application.

3.9. Easier for Implementers

You might think that with all these advantages, using the open data model might be difficult to use for programmers. In fact, we have found that it is *easier* than conventional approaches, primarily because of the higher-level of support that can be provided by tools.

The Macintosh, Motif, Windows, and Java systems all specify how selection handles are supposed to work for manipulating graphics like lines and rectangles (see **Figure 3**). However, *none* of their toolkits support selection handles — each programmer is left to implement them over again. Why? Because selection handles need to know where the graphical objects are and how to move and resize them. With an open data model, the selection handles can be supplied in the toolkit because this information is available. Operations like cut, copy, duplicate, paste, to-top, to-bottom, etc. can also be supplied in

[2] At least none that I could find in Framemaker 5.5 for Windows. Another problem with today's large applications is that there is no easy way to search for capabilities. Maybe with microcomponents, there could be a system-wide way to search for capabilities, like "man –k" for Unix.

a library and often can be used *without change* even for application-specific objects, if there is a standard protocol for determining and manipulating the selected objects. This means that the programmers using the open data model have less to implement.

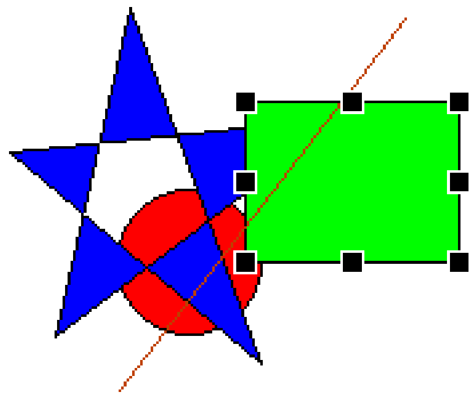

Figure 3. A collection of graphical objects in Amulet. The color of the star can be changed by simply setting its color slot, and Amulet will redraw the star and the other objects that overlap it. The Amulet toolkit includes selection handles (shown here around the rectangle) which support selecting, moving and resizing objects.

Undo is often one of the more difficult operations to implement in applications, especially if "unlimited" undo of multiple operations is required, like in Microsoft Word V6 and after, rather than just single level undo like on the original Macintosh. The open data model makes implementing undo easier since most operations in direct manipulation interfaces involve small, local changes to the data structures, and undoing the operation often is simply a matter of restoring the old value. Since the undo mechanism can query and modify the data structures, and there is a standard protocol for most operations, often undo can be supplied by the library as well. For example, in Amulet, all the standard operations come with their own undo already supplied [19].

Automatic save and load of the application's data structures to files can be provided automatically if there is an open data model because the system can determine the relevant parts of the objects to store and retrieve. As discussed below, the current version of Java supports a partial open data model through the "reflection" mechanism, and also supports save and load, which it calls "serialization."

Many research system (e.g., Garnet [21], Amulet [20], EVAL/vite [7], and SubArctic [8]) and some commercial systems (e.g., Galaxy [27]) have found that *constraints* are a convenient way to implement parts of applications, especially the user interface. A constraint is a relationship that is declared once and maintained by the system. Examples include that a scrollbar must stay on the right of the window or that the wires must stay attached to the nodes even if the nodes are moved. Constraints must be able to access the properties of objects so that the values can be propagated to where they are needed. Thus, using a constraint solver essentially *requires* an open data model so the constraints can access and set the properties of objects.

The open data model also makes debugging much easier. For example, Amulet provides an *inspector* (see **Figure 4**), which can display all the properties of any object. This provides instant object visualization for all objects in all applications. Many users report that this is one of the most helpful features of Amulet. Other systems with advanced debugging features also rely on being able to access the content of objects, such as the "magic lenses" of SubArctic which show the underlying structure and parameters of the graphics [6].

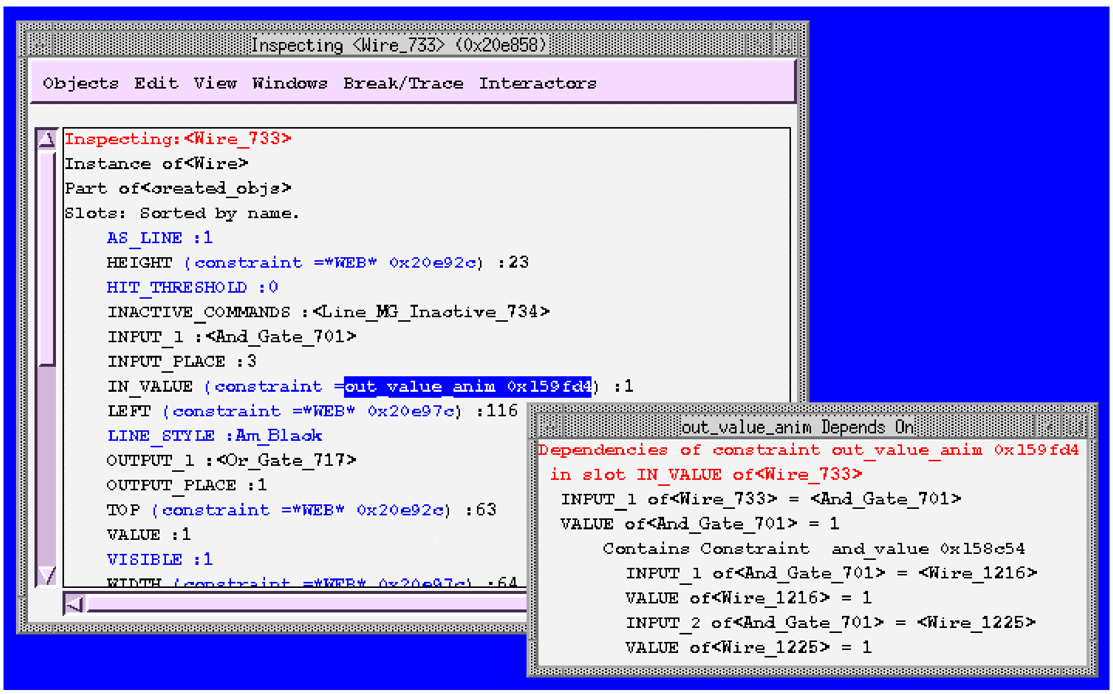

Figure 4. Inspecting a wire object and the constraint in its IN_VALUE slot using Amulet [15]. The open data model allows the inspector to query all the properties of all objects. The slots which are inherited are shown in blue.

4. Some Partially Open Systems

The beginnings of an open model are incorporated into some of today's commercial tools. OLE, AppleScript, Java and MIME are some examples. Even C++ has been retrofitted with some of the required capabilities, with its new "Run-Time Type Information" (RTTI) interface. The inclusion of these facilities gives further evidence as to the importance of the open data model. In some sense, what we are therefore arguing is to continue in the direction that these systems are already moving so that the next generation of tools will support a fully open data model.

Microsoft's OLE and ActiveX provide a protocol called "OLE Automation" where an application can allow external applications to query and change some of their internal state. For example, this can be used to find out the current value of a cell in a spreadsheet. However, many applications do not support this protocol because it is quite complex and it is much harder to implement applications that support it. Furthermore, the ones that do support it only provide access to a small subset of the operations and data.

AppleScript is another technology for providing some access to an application's inner workings. AppleScript is designed so one application can control another by giving commands as if it were the user. However, there are essentially no capabilities for querying the state of the application and for finding out the values of its data structures. Thus, an external program might be able to set a spreadsheet cell to 20, but it could not find out what the current value of a cell is. In particular, AppleScript does not support external programs querying for a meaningful description of objects that are operated upon – just the operation name and an opaque pointer are all that is available. ScriptAgent [12] is a research system that tries to allow users to write scripts using Apple's AppleScript interface. Unfortunately, "several years after Apple introduced AppleScript scripting, and asked application developers to [support it], sadly, few developers have done so. Many applications are at least partially scriptable, but few provide either a complete scripting interface or a usable recording capability" [12, p. 42].

The OpenDoc environment [1] from Apple (now canceled) was not going to improve on this much. It contained a "services" protocol that was supposed to allow external programs to access the application's data, but this part of the OpenDoc specification was apparently never finished.

The problem is that implementing a scriptable and examinable interface in these systems is extra work for programmers since it must be done *in addition* to the application's functionality and user interface. In contrast, the open data model is *easier* for programmers than conventional designs and the examinability and scriptability come for free.

Java has taken a big step towards an open data model with the its "reflection" interface, which allows properties of objects to be queried at run time. The Java Beans architecture also defines a set of naming conventions that allow many properties of components (called "beans") to be discovered by their containers (called "bean boxes"). This provides many advantages to the Java programmer, including automatic support for "serialization" (saving to and loading from files) and a much simpler component model.

On the Internet, most of the data is currently stored in an open format. As mentioned above, HTML files are examinable by any program. The Multipurpose Internet Mail Extensions (MIME) standard supports dynamically tagged data in electronic mail and World-Wide Web pages. The MIME types are easily inspected by people or programs, and can be discovered at run-time. MIME defines some basic types of data that are popular and universal, and makes it very easy to add new types as needed.

5. Amulet's Open Data Model

The Amulet user interface development environment [20] incorporates many of the features of the open data model. It is being developed as part of a research project at CMU, and is distributed for free.[3] Amulet supports the creation of user interfaces in C++ for Unix, Microsoft Windows 95 and NT, and the Macintosh. Amulet is downloaded over 200 times a week (over 11,000 times in the last year alone), and many research and commercial systems have been created with it, so many people have experimented with the features described here. Amulet's predecessor, called Garnet [21], was in Lisp and was also used by many people. The best ideas from Garnet were carried forward into Amulet, so the following focuses on the open data aspects of Amulet.

The main ways that Amulet supports the open data model is with an open and flexible object model, and with inspectable and manipulable behavior objects.

5.1. Open Data Objects

Similar to Dertouzos's "e-forms," all objects in Amulet are implemented as attribute-value pairs. Amulet's objects use a prototype instance object system implemented on top of the C++ object system. Each object is a collection of named "slots," which correspond to instance variables in other object systems. The attributes are the slot names, and the values are the data in the slots. Unlike conventional *class*-instance systems such as C++ or SmallTalk, in a *prototype*-instance object system there is no distinction between classes and instances. Any object can serve as a "prototype" for new instances, and instances immediately inherit the slots of the prototype as the default values. Local values can then be assigned to any slots. Another interesting feature in the Amulet design is that there is no difference between method slots and data slots: a method is simply a type of data, and can be local or inherited in any object. Slots in Amulet can be

[3]Amulet is in the public domain. You can download the source and documentation from http://www.cs.cmu.edu/~amulet or send mail to amulet@cs.cmu.edu.

dynamically created at run time and they are dynamically typed, so the same slot can hold an integer at one time and a string later.

A key feature of the object system is that the full information about an object is examinable at run time. A program can query the type of an object (which corresponds to what its prototype is). Also available is the list of the slots currently in the object along with the type and data of the value in each slot. Method slots can be queried as easily as the data slots. **Figure 4** shows the inspector viewing the complete set of slots of an object. Other "meta" information about the slots is also available, such as whether slot is read-only, what kind of inheritance has been defined for the slot, etc. (see Amulet's reference manual [17] for complete details).

Often, objects will contain a list of other objects as components. For example, a window object will contain a list of graphical objects that are in the window. Standard methods are available to querying the component objects that are part of an object. This allows external applications to find all the objects in an interface. Since widgets and custom graphics are all implemented with Amulet objects, the complete contents of the user interface is examinable.

Amulet establishes a set of conventions about the names of slots used for different parameters to objects. All graphical objects must export their bounding box in slots called LEFT, TOP, WIDTH, and HEIGHT which must have numeric values. Setting these slots adjusts the object. This makes it possible for Amulet to supply a selection handles widget, which in other toolkits must be re-implemented by each application (see Figure 3). Other standard slots control the color and visibility of objects, and these can be read and set. The VALUE slot is used by all objects which compute or allow the user to enter a value. In a scroll bar widget, the VALUE slot holds a float or an integer to specify the percent of the way up or down. In a text input widget, the VALUE slot will hold the current input text string. Setting the VALUE slot changes the value of the widget. The ACTIVE slot is used to control all objects that can be disabled, so setting the ACTIVE slot to false will cause a widget to "gray out" so it cannot be operated. All widgets that are labeled use the slot called LABEL. Of course, each kind of object will add additional slots with specific meanings, but a program can count on some standard slots always being available in objects of a particular kind.

The data put into slots in Amulet is "self-describing," since the types of all values are available at run time. There are well-known type descriptors for all of the built-in types, such as integers, strings and floats, and we have added some other useful types such as various forms of dynamic lists. Applications are free to create their own new custom types, although this is often not necessary since many application-specific data structures can be created just using Amulet objects and lists.

Amulet uses *dynamic typing* since a single slot can hold any type of value. A "boxed type" called `Am_Value` is available for C++ programs that want to hold a value of any type, and still be able to query the type and perform type-specific operations. As argued by John Ousterhout, dynamically typed languages:

> "are just as safe as system programming languages… [Dynamically typed languages] do their type-checking at the last possible moment, when a value is needed. Strong typing allows errors to be detected at compile time, so the cost of run-time checks is avoided. However, the price to be paid for efficiency is restrictions on how the information can be used; this results in more code and less flexible programs" [24, p. 26].

The result of using the open data model in Amulet is that it is very easy to iterate through all the objects in a window, and find out the types of each object, querying and modifying appropriate slots. Dialog boxes are just windows with a "modal" bit set, so the same iterators work to find the widgets in a dialog box. A program can easily find, for example, all the text objects or all the widgets in another program, and get or set their current values. The generalized searching (**Figure 1**) and inspecting (**Figure 4**) facilities discussed above are therefore be easy to attach to any interface created using Amulet.

Like Java, Amulet has a built-in save and load mechanism that can be used by all objects. The saved file is in a standard format, where the types of the values are tagged, so a generalized search mechanism would be possible that could search in disk files for graphics as well as text.

Amulet's open data model allows it to supply constraints to help with the implementation of user interfaces. Any slot of an object can contain a constraint instead of a constant value. The constraint can contain arbitrary C++ code, and can return a value of any type (so they are not restricted to being used just for numbers). Constraints can access any value from any object, since all values are stored in a uniform manner. Whenever any of the slots change that the constraint depends on, the constraint is reevaluated, so the slot that the constraint is in will receive an updated value.

5.2. Open Behaviors

In addition to helping with the static graphics that are part of an application, the open data model also helps with the dynamic *behavior* of the interface. The behavior is how the objects respond to the user's input events. In addition to *scriptability*—the ability to observe, record and replay the user's actions, Amulet also provides *examinability* for the behaviors themselves—the ability to query an application to find out what the end user can *potentially* do. This enables a wide range of new capabilities that are not provided by other toolkits, such as customization, intelligent help, and generalization of scripts.

There are two important aspects to making the behavior open: access to what the user *has done*, and access to what the user *can do*. Some systems offer access to what the user has done through scripting mechanisms, such as AppleScript. Very few systems allow the investigation of what operations are available to the user. Note that we are not talking about access to the low-level input events (mouse left button down at (102, 45), or control–v key hit), but rather a high-level description of the resulting operation and the parameters to the operation.

Amulet supplies both kinds of access through the use of standard objects to represent all behaviors. All high-level data operations are encapsulated into "Command" objects, and low-level, direct manipulation interaction techniques are handled by "Interactor" objects. The open data model enables these objects to work, and also enables the behavior objects themselves to be investigated. Command objects for operations that have been executed are available on the undo history list, and all potential commands and Interactors of an application can be found by investigating the objects attached to the application's windows.

In Amulet, all high-level operations are encapsulated into "command objects" [19]. In many other toolkits, when a widget like a button is used, the programmer must write a "call back procedure" (in Motif) or an "event method" (in Visual Basic) to handle the behavior. In Amulet, instead a command object is allocated, and its DO method is called. One result of the open data model is that these command objects can be provided in the library and used by many applications without change. Each command object contains slots and methods to perform the operation, and also to handle undo, selective undo and repeat, enabling and disabling the command (graying it out), and help messages. Thus, unlike other uses of command objects, such as in MacApp [28] (where the information about when a command is available is not programmed as a method of the command), the command objects in Amulet provide a single place for describing all of a behavior.

The Amulet library contains built-in commands for move-object, create-object, change-property, become-selected, cut, copy, paste, duplicate, quit, to-top and to-bottom, group and ungroup, undo and redo, and drag-and-drop. In systems that do not have an open data model, these operations must always be re-implemented by each application. As with graphical objects, command objects observe a convention about which slots are used, so that the effects of a command are inspectable. The objects which are modified by a command are stored into the OBJECTS_MODIFIED slot of the command object, and values are stored in the VALUE and OLD_VALUE slots. For example, this enables the undo history to display useful information about each command in addition to the command name (which is all that can be displayed by most other systems).

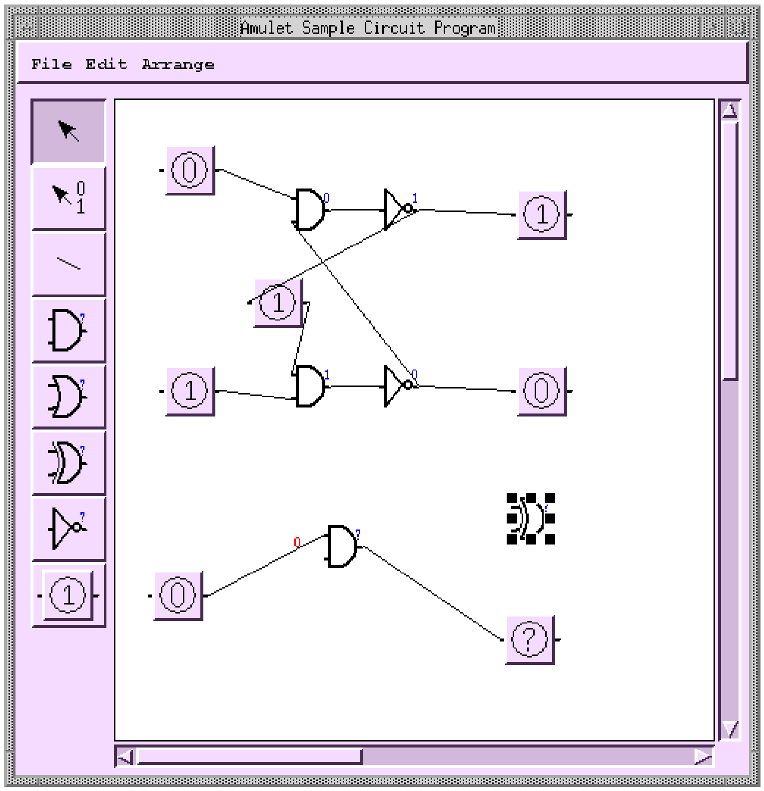

Figure 5. A simple circuit design program created with Amulet that supports full editing operations (cut, copy, paste, moving with the selection handles, etc.), load and save, animation (the red 0 is moving along the bottom left wire), and gesture recognition (drawing an "O" with the right mouse button down creates an OR gate, drawing an "A" makes an AND gate, etc., so the user doesn't have to keep going back to the palette on the left). Scripting of operations is also supported. Due to Amulet's use of the open data model and other high-level features, this entire application only requires about 850 lines of C++.

It is worth emphasizing that the open data model, in addition to making it possible for the command objects to be used without change, also enables automatic support for undo in many cases. Whenever programmers use the standard command objects, all operations are automatically undoable without writing any extra code. If the programmer creates custom commands that perform application-specific actions, then a custom undo method will have to be written as well. For example, in a circuit program (see **Figure 5**), deleting the gates is handled automatically (along with the Undo that puts them back), but the programmer has to write a method to delete the attached wires when a gate is deleted, and also a corresponding undo method that restores the wires. However, we have found that the open data model makes writing undo methods very easy. Most operations are implemented by modifying slots of the objects, so the DO method simply copies the old values of the object's slots into the OLD_VALUE slot of the command object before the modification, and the Undo method simply restores the saved values. Usually the application's data structures do not need to be modified to support undo.

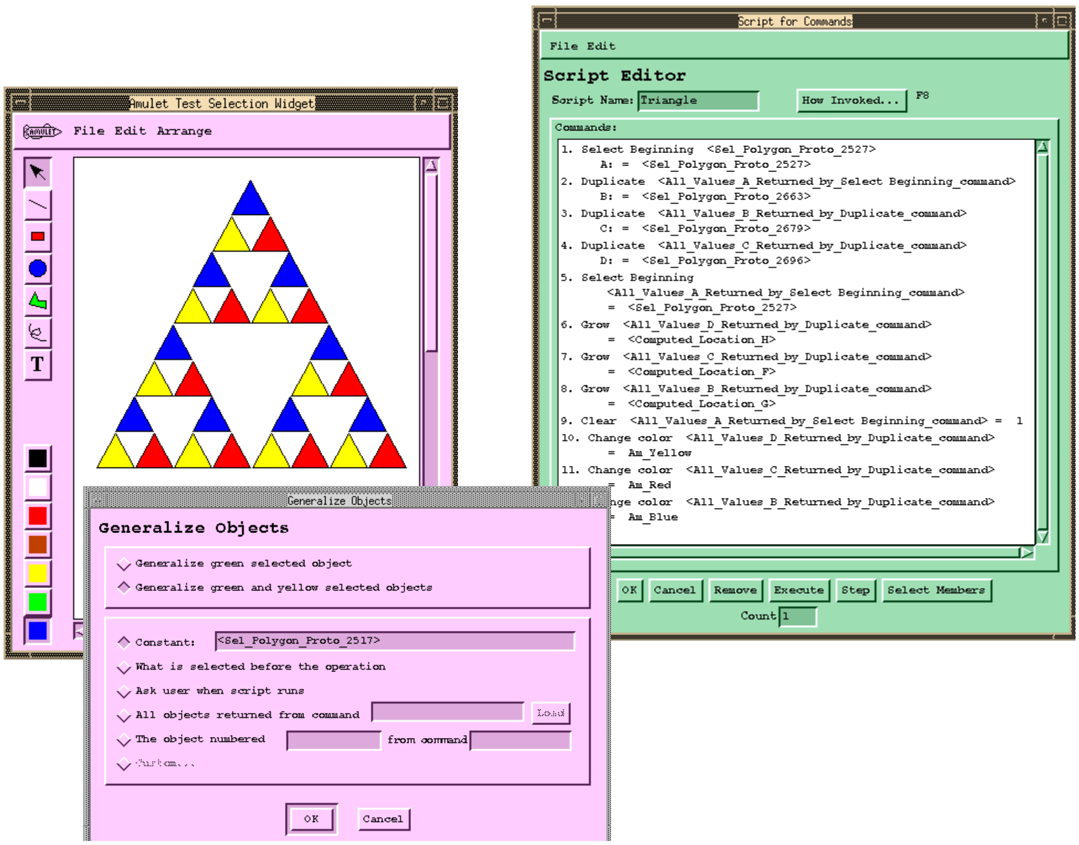

Figure 6. On the left is a drawing program created using Amulet showing the result of a script which subdivides a triangle into 3 smaller triangles, applied 13 times. This is called a "Sierpinski Gasket." In the upper right is the script, and the lower window shows the dialog box for generalizing references to objects.

The inspectable nature of the commands also supports scriptability. Amulet's scripting mechanism [16] copies commands from the undo history for use in the new script. The open data model allows the scripting mechanism to be independent of the original application—the parameters of the command can be inspected and generalized. As shown in Figure 6, Amulet can detect references to objects, locations, and other values in the scripts. In some cases, it can automatically generalize them. For example, when an object is created during the script and subsequently modified, then when the script is run, the newly created object will be modified rather than the original example object. The user can also specify how to generalize or edit the parameters. The important point is that the application (in this case, the drawing program) does not need to write extra code to support scripting since the open data model gives the scripting facility sufficient power to inspect the command objects and manipulate their values. When the script is to be re-executed, the appropriate method of each command can simply be called.

The open data model also enables behaviors such as the standard direct manipulation interaction techniques to be supplied in the Amulet library. In most other toolkits, when application-specific graphical objects are to be manipulated by the user (for example, to move a rectangle with the mouse), the programmer must write methods on the graphical object itself to handle the mouse down and mouse move events. In Amulet, an entirely different design is used. Built-in behavior objects, called *Interactor objects*, are attached to the graphical objects, and the Interactor objects handle all the input [14]. There are six basic kinds of Interactors, and most direct manipulation behaviors can be achieved by simply creating an instance of a built-in Interactor and setting a few parameters. For example, to move a rectangle, a Move-Grow-Interactor is attached to it, and this Interactor waits for a down press on the rectangle, and handles moving the rectangle until the release. An important advantage of the Interactor model is increased reusability since common options like gridding, resizing objects from any side, dealing with minimum sizes, allowing interactions to be aborted, etc., are all implemented as part of the built-in Move-Grow-Interactor and can be enabled as needed. When the Interactor is finished, it allocates a command object so the operation can be undone. Other Interactor types in Amulet handle selecting one or more from a set of objects, creating new objects, text editing, gesture recognition, etc.

The open data model allows Amulet to successfully separate the “Controller” from the “View” in the “Model-View-Controller” idea from Smalltalk [10]. This is because the open data model allows the Interactors (as the controllers) to search for graphical objects (as the view), and always know how to manipulate the graphical objects, even if the objects are custom and application-specific. Most previous systems, including the original Smalltalk implementation, had the View and Controller tightly linked, in that the controller would have to be re-implemented whenever the view was changed, and vice versa. Indeed, many later systems such as Andrew [25] and InterViews [13] combined the view and controller and called both the “View.” In contrast, the open data model allows Amulet’s Interactors to be independent of graphics, and the Interactors can be reused in many different contexts.

The ability to find and examine the behavior objects themselves allows external utilities to find out what the end user can potentially do in an application. This is possible in

Amulet because each behavior is represented by an explicit Interactor object or command object. For example, a customization facility can be independent from the application because it can query to find out all the operations that are available. The help system and intelligent agents can find where particular commands are located on the menus or dialog boxes so these can be highlighted. The scripting mechanism can determine all the ways there are to set a value, and provide these as options for generalizations. For example, if the user writes a script to change the color of objects, the scripting mechanism can query the widgets of the application to find out which ones change the color of objects, and provide these as options for ways to choose the color when the script is run. Similarly, agents, help systems and tutors in Amulet would have no problem executing commands as if they were the user, since the appropriate command object can be found and its DO method executed. The debugging facility in Amulet can provide facilities to investigate why behaviors did or did not execute by examining the events that trigger them and the objects they operate on. This allows the Inspector to answer questions such as “why did this Interactor not run when I clicked the left button” or “why is this command not active now” [15].

6. Concerns About the Open Data Model

The main concerns with the open data model are that it would be too expensive to implement systems using it, that it does not provide sufficient information hiding, and that it is politically and technically difficult to achieve.

As computers continue to get inexorably faster with bigger memory and bigger disks, most of the new power is being devoted to the user interface. Yesterday’s computers were already able in real-time to format documents, recalculate spreadsheets, and maintain constraints in drawings. No new computing power is needed for the basic functionality used by the vast majority of computer users. Yesterday’s disks were already big enough to hold all the text that a person might read or create in a lifetime, so storing files in a slightly less efficient format will not have much impact. Instead, we can afford to use the increased power of computers to make the users’ and programmers’ lives easier. Since the open data model has so many benefits for both users and programmers, it is worth spending the extra resources needed by it.

The most serious issue with the open data model is information hiding and security. How can an application’s implementation details be kept hidden? Whereas since it is a research system, Amulet provides a completely open environment that allows anything to inspected and modified, Java shows how an open model can be more secure. In Java, properties of objects can be declared public or private, and the reflection capabilities can be limited to only the public properties. Modifiability of the properties can be allowed only through the object's public methods.

A related question is that if all applications are required to publish their data formats, then what happens when a new version needs new fields of the data structures, or if old fields disappear or change types? Part of the answer is that external applications do not need to be hard-wired with knowledge of a data structure's format because they can inquire as to the properties of objects. The Amulet architecture tags the data with types and other meta-information, which allows external applications to determine the types of data at run time. Java's "reflection" mechanism supports the same kind of inspection of an application's data structures. Thus, the external applications will not necessarily need to be recoded when the data structure evolves.

Can this vision be achieved? Technically, Amulet demonstrates that many of the desirable characteristics can be implemented fairly easily, and section 3.9 argued that using the open data model will be *easier* for programmers than today's protocols. Politically, all applications have been willing to adhere to some previous standards, if there are clear benefits and they do not impose too much extra work. For example, Apple stipulated that all applications for the Macintosh must support a standard set of formats for the clipboard for Cut and Paste, including a textual format and a picture format. Applications are free to add additional complexity if desired. Every Macintosh application supports the clipboard. On the PC, Microsoft's OLE integration capabilities are being used by at least some of the Windows software, and would arguably be used by many more if OLE was not so complex to adhere to. Building the capabilities into the programming language, as in Java, makes the open data model even easier to support. If a powerful vendor or consortium was to endorse the open data model, then end users *and programmers* would benefit, so we can hope that the industry will continue to move in this direction.

7. Conclusions

We are excited by the many possibilities enabled by the open data model. A number of researchers are exploring this model, and some open data capabilities are becoming available in new commercial systems like Java.

We hope that a full open data model will soon be available in commercial toolkits, so that all of the advantages can be widely exploited. These include the built-in, ubiquitous support for automation, macros and end user customization, agents and tutors, advanced checkers, rich search and replace, semantic markings, alternative interfaces, and micro-components. At the same time, the open data model makes it easier for the programmers because more higher-level capabilities can be supplied by the toolkit and reused by applications. For example, the Amulet toolkit demonstrates that selection handles, undo, editing commands, scripting, searching, file save and load, and debugging can be supplied in a library and used without changes by many applications. The result of using the open data model is more features for users while at same time making it easier for programmers, so we hope that it will be increasingly studies and adopted.

Acknowledgments

For help with this paper, I would like to thank Dan Olsen, Brad Vander Zanden, and Rob Miller.

The Amulet toolkit has been developed by Brad A. Myers, Rich McDaniel, Rob Miller, Alan Ferrency, Andrew Faulring, Ellen Borison, Bruce Kyle, Andy Mickish, Alex Klimovitski, and Patrick Doane.

References

1. Curbow, D., *et al.*, *Human Interface Specification for the Macintosh Implementation.* Apple Computer, Inc., OpenDoc Version 1.0, Specification Version 1.0.3, 1995, http://www.opendoc.apple.com/.

2. Cypher, A. “EAGER: Programming Repetitive Tasks by Example,” in *Proceedings SIGCHI'91: Human Factors in Computing Systems.* 1991. New Orleans, LA: pp. 33-39.

3. Dertouzos, M., *What Will Be.* 1997, San Francisco: HarperEdge.

4. Dragon Systems Inc., “DragonDictate v3.0,” 1998. http://www.dragondictate.com/.

5. Greenberg, S., Gutwin, C., and Roseman, M. “Semantic Telepointers for Groupware,” in *OzCHI '96 Sixth Australian Conference on Computer-Human Interaction.* 1996. Hamilton, New Zealand: pp. 24-27.

6. Hudson, S., Rodenstein, R., and Smith, I. “Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging,” in *Proceedings UIST'97: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1997. Banff, Alberta, Canada: pp. 179-187.

7. Hudson, S.E., *A System for Efficient and Flexible One-Way Constraint Evaluation in C++.* Graphics Visualization and Usability Center, College of Computing, Georgia Institute of Technology, 1993,

8. Hudson, S.E. and Smith, I. “Ultra-Lightweight Constraints,” in *Proceedings UIST'96: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1996. Seattle, WA: pp. 147-155. http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

9. IBM, “ViaVoice,” 1998. http://www.software.ibm.com/is/voicetype/.

10. Krasner, G.E. and Pope, S.T., “A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system.” *Journal of Object Oriented Programming*, 1988. **1**(3): pp. 26-49.

11. Kurlander, D. and Bier, E.A. “Graphical Search and Replace,” in *Proceedings SIGGRAPH'88: Computer Graphics.* 1988. Atlanta, GA: **22**. pp. 113-120.

12. Lieberman, H. “Integrating User Interface Agents with Conventional Applications,” in *1998 International Conference On Intelligent User Interfaces.* 1998. San Francisco, CA: pp. 39-46.

13. Linton, M.A., Vlissides, J.M., and Calder, P.R., “Composing user interfaces with InterViews.” *IEEE Computer*, 1989. **22**(2): pp. 8-22.

14. Myers, B.A., “A New Model for Handling Input.” *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 289-320.

15. Myers, B.A., “Debugging 'Why-Not' Questions in Graphical User Interface Software,” 1998. Submitted for Publication.

16. Myers, B.A. “Scripting Graphical Applications by Demonstration,” in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 534-541.

17. Myers, B.A., *et al.*, *The Amulet V3.0 Reference Manual.* Carnegie Mellon University Computer Science Department, CMU-CS-95-166-R2, 1997, http://www.cs.cmu.edu/afs/cs/project/amulet/amulet3/manual/Amulet_ManualTOC.doc.html.

18. Myers, B.A., *et al.*, “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces.” 1990. **23**(11): pp. 71-85.

19. Myers, B.A. and Kosbie, D. “Reusable Hierarchical Command Objects,” in *Proceedings CHI'96: Human Factors in Computing Systems.* 1996. Vancouver, BC, Canada: pp. 260-267.

20. Myers, B.A., *et al.*, “The Amulet Environment: New Models for Effective User Interface Software Development.” *IEEE Transactions on Software Engineering*, 1997. **23**(6): pp. 347-365. June.

21. Myers, B.A. and Vander Zanden, B., “Environment for Rapid Creation of Interactive Design Tools.” *The Visual Computer; International Journal of Computer Graphics*, 1992. **8**(2): pp. 94-116.

22. Mynatt, E. and Edwards, W.K. “Mapping GUIs to Auditory Interfaces,” in *Proceedings UIST'92: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1992. Monterey, CA: pp. 61-70.

23. Olsen Jr., D.R., *et al.* “Generalized Pointing: Enabling Multiagent Interaction,” in *Proceedings SIGCHI'98: Human Factors in Computing Systems.* 1998. Los Angeles, CA: pp. 526-533.

24. Ousterhout, J.K., “Scripting: Higher-Level Programming for the 21st Century.” *IEEE Computer*, 1998. **31**(3): pp. 23-30. March.

25. Palay, A.J., *et al.* “The Andrew Toolkit - An Overview,” in *Proceedings Winter Usenix Technical Conference.* 1988. Dallas, Tex: pp. 9-21.

26. Raskin, J., “OpenDoc: Another Half a Loaf.” *Interactions*, 1997. **4**(3): pp. 14-15. May+June.

27. Visix Software Inc., “Galaxy Application Environment. 11440 Commerce Park Drive, Reston VA 22091. (800) 832-8668,” 1997. Company dissolved in 1998.

28. Wilson, D., *Programming with MacApp.* 1990, Reading, MA: Addison-Wesley Publishing Company.