Tcl and Tk Use in the Artifact Based Collaboration System

John Menges
Mark Parris[1]

May 24, 1993

Abstract

At the University of North Carolina Department of Computer Science we are developing the Artifact Based Collaboration System (ABC) which provides computer support for collaboration. ABC consists of a distributed hypermedia graph server with associated graph browsers. It also includes the ABC Matrix which interconnects the graph server and browsers with existing X-based applications. ABC also provides hyperlinking and window sharing capabilities. Tcl is used for defining process-level interfaces and for inter-application communication. Tk is used to implement various user interfaces. In this paper we discuss current and expected uses for Tcl and Tk in the ABC system, and some of the problems we have encountered regarding the use of Tcl and Tk in our environment.

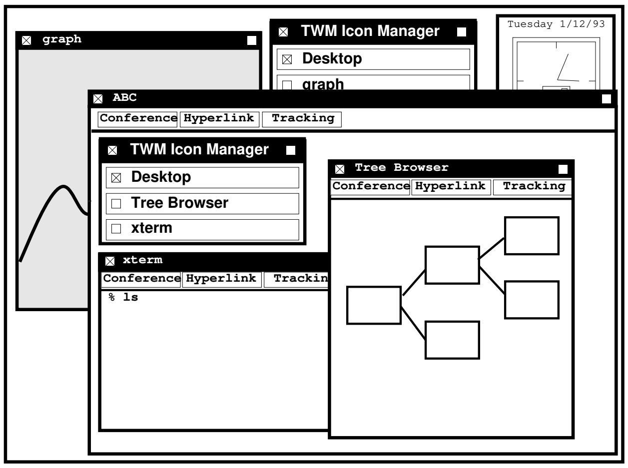

Figure 1: An ABC Virtual Screen

1 Introduction

At the UNC Department of Computer Science we are developing a system to provide computer support for collaboration. The system is called the ABC (Artifact Based Collaboration) System[5][1] and runs on UNIX[2] under the X Window System[3]. ABC supports manipulation and sharing of the artifacts which are the products of group collaboration. It also provides for recording and studying the way the system is used. The artifacts include, but are not limited to, documents, diagrams, and code as well as graphs which represent relationships among these items. These artifacts are the objects of collaboration among groups of people who may or may not be co-located in either time or space.

Below, we first describe the functionality and some of the appearance of the system. Then we discuss a high-level view of the design of the system. Finally, we discuss our use of Tcl and Tk in the implementation of ABC.

2 Appearance of the ABC System

The ABC system is not a single application, but an environment in which applications may run. The boundary between this environment and the normal UNIX environment is represented on the the user's display as a virtual root window. This window exists as a different logical display to which X-clients running in the ABC system may connect. This window also has its own window manager (Figure 1).

In the ABC system, artifacts are represented as nodes and node content in graphs. Figure 2 shows a sample graph with four nodes. The leaf nodes (nodes A, B, and C) each have content. The content of each node is shown in the rectangles attached by dashed lines. The graph artifacts can be manipulated by hypermedia browsers that are part of the ABC system itself, while the other types of content can be manipulated by arbitrary X-based applications that are not aware

[1] John Menges (menges@cs.unc.edu) and Mark Parris (parris@cs.unc.edu) are graduate students in Computer Science at the University of North Carolina.

[2] UNIX is a registered trademark of AT&T.

[3] The X Window System is a trademark of the Massachusetts Institute of Technology.

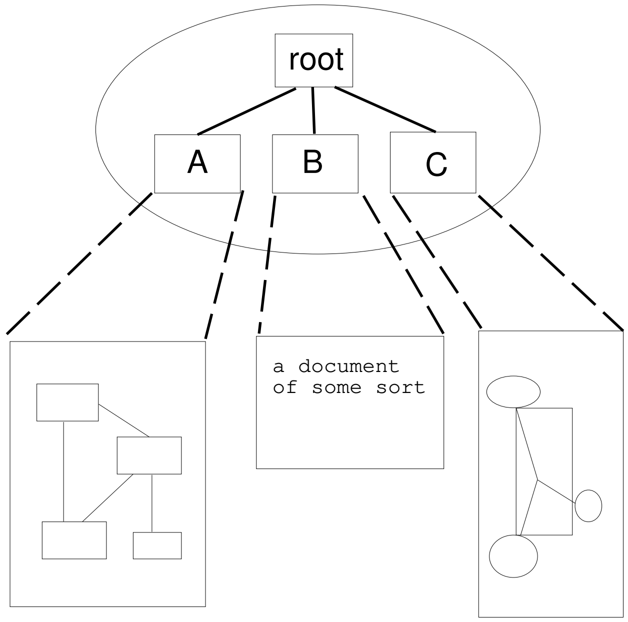

Figure 2: A Sample Graph

of the ABC system or its organization of the artifacts. Nodes may also be joined by hyperlinks, either with endpoints which are simply nodes or endpoints which are anchors within the content of nodes. These anchors can be words, sentences, or parts of figures or graphs.

Sharing of these artifacts can take two forms. First, artifacts may be shared asynchronously by accessing shared artifacts through the distributed graph server (DGS)[4]. Artifacts may also be shared synchronously by sharing windows which display the artifacts. In this case two users see the same windows encapsulated in a nested Virtual Screen[2] in a WYSIWIS (what-you-see-is-what-I-see) fashion. Input to the windows is arbitrated by passing a token to the next user who wishes to provide input.

In addition to these functions available to the normal user we also incorporate facilities for monitoring and replaying sessions. These facilities are used by researchers to study the way that groups use the system. These functions include simple recording and replay of all of the windows in the user's ABC environment as well as recording semantic actions in various applications.

Finally, the system provides an interface to the ABC functions which can be used both with ABC built applications and standard X applications.

3 Design of the ABC System

The ABC system has five principle components. They are the distributed graph server (DGS), browsers, the generic function manager, the X Pseudo-Server, and the matrix hub. Additionally, the system is designed to allow standard X applications to be used either without modification or with minimal modification to add features such as anchored hyperlinks. Each of these items is discussed below.

All of the artifacts used in the ABC system are stored in a distributed graph server (DGS) built as part of the ABC system. The DGS represents all artifacts as objects (nodes, subgraphs, or node content) in a large graph. Browsers are used to manipulate and traverse this graph structure, providing specialized tools for manipulating lists, trees, and networks.

The generic function manager (GFM) acts as an interface to the ABC system. Since many of the functions of the ABC system operate on a particular window (or on a node represented by a window) this interface exists for each window in the ABC system. To provide this interface we add a bar between top-level X windows and the window-manager title bar as shown on the ABC, Tree Browser, and xterm windows in Figure 1. Added functions are invoked via this function bar, which is managed by the GFM, like a window manager manages the title bars it creates. The GFM provides the user interface, but another process, the matrix hub, keeps track of what's going on in the system. By recording which window is open on which node the matrix hub, working in conjunction with the GFM, is able to associate requests such as 'create hyperlink' with the artifacts that are being examined in the respective windows. In turn, the matrix hub communicates this link to the graph-server. The GFM bar is also used to invoke conferencing operations, such as 'move this window into a conference', and other system operations.

In addition to the browsers built for the ABC system, we use standard applications to manipulate node content. Without any modifications these applications can be used to edit and manipulate node content, including creating and following un-anchored hyperlinks (by means of the GFM bar). Creating or following anchored hyperlinks does require some minimal modifications to the application. A modified application must provide a mechanism for identifying an item or region designated as an anchor and passing that information to a library function. This function will send a message to the GFM informing it of the selected region so that the matrix hub may record the link. Applications may also be modified similarly to record what semantic actions were invoked.

Finally, we have the component which manipulates the X protocol stream. Windows are grouped, conferenced, and tracked by using an X Pseudo-Server (XPS) which can intercept, translate, and distribute X-protocol streams[3]. The XPS can modify the X-protocol streams for a set of windows so that they all appear to have the same (virtual) root window, thus grouping them inside a common Virtual Screen. It can also conference windows by distributing and translating the X-protocol stream for a particular window to several other XPSs associated with the other conference participants. Tracking is accomplished by recording summaries of each message that passes through the protocol server along with time-stamp information.

All of these components function together to provide an environment in which groups can work on shared artifacts. Below we discuss some of the ways that we use Tcl and Tk in the implementation of these components.

4 Using Tcl and Tk with C++

Tcl and Tk were designed to be used with the C programming language, but the ABC system is being written in C++. Fortunately, C++ is able to call C code and vice versa, so it is possible to use Tcl/Tk with C++. However, using Tcl/Tk effectively and naturally in a C++ environment poses certain problems. In this section we discuss the problems we have encountered and the solutions we have developed to date.

4.1 A Tcl/Tk Interpreter Class

It is possible to call Tcl/Tk C library routines directly from C++ code. It is preferable, however, to encapsulate the Tcl/Tk C library calls in one or more C++ classes. For example, one might have a Tcl/Tk interpreter class whose constructor creates a Tcl interpreter and whose member functions map directly to Tcl/Tk C library calls. Extended Tcl does this type of encapsulation for Tcl (but not Tk) via the tcl++.h include file.

The primary advantage of this approach is that it enables C++ code to make Tcl/Tk library calls in a style that fits the object-oriented nature of the C++ language. For example, to create an interpreter, one would create an instance of the interpreter class, and the constructor for the class will automatically call the **Tcl_CreateInterp** library routine to create a Tcl interpreter. The constructor can also do any initialization of the Tcl interpreter, e.g., by adding procs or command bindings to C routines that implement commands. Tcl/Tk library routines such as **Tcl_CreateCommand** which take a Tcl interpreter as their first argument can then be implemented as member functions in the interpreter class, their signatures being the same as the corresponding Tcl library routines, sans the interpreter argument. C++ features such as default arguments and function overloading can now be applied to the Tcl interface, just as they can be applied to the other classes in the application. Furthermore, we can add our own member functions to the interpreter class (ones that don't map directly to Tcl/Tk library routines) to enhance the capabilities of our Tcl interpreter.

4.2 Callbacks to C++ Code

Tcl and Tk have mechanisms for calling C procedures from the Tcl interpreter. These C procedures are called *callbacks*. For example, Tcl commands can be implemented in C, and the C callback routines implementing these commands can be *registered* with the Tcl interpreter using the **Tcl_CreateCommand** routine. Registering a command with the interpreter associates the callback routine with a command name. When the command is interpreted, the corresponding C callback routine is called.

In C++ code, we can use the "extern C" declaration to define C++ routines with C calling conventions that can be used as callback routines called by the Tcl interpreter. For the C++ programmer, however, it is often desirable to have the interpreter call particular member functions of particular objects.

We have developed a strategy for implementing Tcl callbacks to member functions of particular objects. Since it is difficult to describe our strategy and its justification in a small space, we have included a detailed description as an appendix to this paper.

4.3 Shadow Variables

It is often necessary to have C++ variables (usually members of objects) "shadow" Tcl variable values, and vice versa. Building on our technique for implementing callbacks, we are implementing C++ classes whose objects act like regular C++ data types (either built-in or user-defined types), but which keep their values consistent with corresponding Tcl (string) variables.

For example, one might declare an shadow integer as follows:

```
Tcl_int speed(interp, ''speed'');
```

The **Tcl_int** class implements an **operator int()** conversion operator and assignment operators, enabling **speed** to be used in any way that a normal **int** variable can be used. The assignment operators, however, call **SetVar** to update the Tcl "speed" variable, and the **Tcl_int** class has a member function which has is registered (in the constructor) to receive notification when the Tcl "speed" variable changes. The **Tcl_int** object then updates its value accordingly.

More complex user-defined data types can also be shadowed. For example, we will implement a bitvector type in C++ which shadows a Tcl type (perhaps an array) which is manipulated by a Tk checkbox list. This makes it possible for the C++ bitvector to automatically reflect the current on and off values of the various checkboxes in the list.

5 Non-blocking Tcl IPC

Our X Pseudo Server (XPS) and other components of the ABC system must not block on any operation, including the sending and receiving of Tcl commands. With the Tk-based **send** mechanism and similar IPC capabilities available via socket-based Tcl extensions, processes on either the sending or the receiving end can block. We are using Tcl-DP[4] RPC as a basis for our Tcl IPC, and are re-implementing the receiving end of RPC and the sending end of RDO to ensure non-blocking operation. Since the XPS and other components have their own schedulers, and the XPS doesn't need Tk at all, we are also modifying RPC and RDO to work with our scheduler and to eliminate any dependence on Tk.

6 Scheduling

Our current browsers use Interviews and ISIS. Existing applications may use other toolkits, and in any event, have their own event loop. As previously mentioned, our X Pseudo-server has its own scheduler. Often different components of applications want to do their own event scheduling. We are still studying this problem and are developing techniques for integrating Tcl and Tk into such environments.

7 New Tk Widgets

We are considering implementing two new Tk container widgets. One would be a Virtual Screen widget which would be able to contain arbitrary top-level X windows of other applications. This would be used in implementing our Virtual Screen window sharing capabilities. The other would be a Generic Function Manager (GFM) widget, which would be able to reparent a single existing top-level X window, adding a title bar at the top like a window manager (but under any existing window manager title bar). Arbitrary Tk widgets such as buttons and menu buttons could be placed on the title bar portion of the GFM widget. These would be used to invoke, e.g., hyperlinking and conferencing functions.

If and when we implement the above-mentioned GFM widget, it will take us a long way toward the goal of having a Tk-based (and therefore highly extensible) window manager.

8 New Tk Geometry Managers

Our graph browsers, currently written using Interviews, implement various constraints on the graph structures they will create. For example, we have Tree browsers, List browsers, and Network browsers. We are considering re-implementing browsers as Tk geometry managers, so that the constraints and graph layouts can be implemented in terms of geometry management. Widgets would be placed in terms of their (logical) link relationships to other widgets, rather than in terms of physical layout relationships to othe widgets. For example, a Tree browser geometry manager would accept placement specifications consisting of parent/child relationships between the widgets. The geometry manager would then automatically compute a reasonable physical layout for the contained widgets.

[4] Tcl-DP was developed by Lawrence Rowe, Brian Smith, and Steve Yen at the University of California at Berkeley

9 Security

The Tcl interpreters we use cannot be allowed to execute arbitrary commands received from arbitrary processes, because some of the built-in Tcl commands (e.g., exec) are unsafe. There are two good approaches to this problem.

One approach is to authenticate connections using an authentication server. This way, we know who is connecting to us and can determine whether or not to interpret commands from the sending process.

The other approach is to have two Tcl interpreters in our applications. One is made safe by removing (renaming to null) any unsafe commands. The other is a full-blown Tcl interpreter. Any commands received from another process are sent to the safe interpreter. If the services of the full-blown Tcl interpreter are needed either internally or for the interpretation of a command received from another process, the full interpreter can then be called upon from inside the application owning the interpeter, in a safe manner.

In the near term, we will be using the latter approach, because it is easier to implement. The former approach allows for multiple classes of requesters which are trusted to different degrees, should that be desirable at some point in the future.

10 Summary

In summary, the ABC System is a large, complex system that makes heavy use of Tcl and Tk for its infrastructure and user interfaces. ABC requires that Tcl and Tk be used in novel ways, and we believe the solutions to our requirements will be generally useful in other environments.

References

[1] K. Jeffay, J.K. Lin, J. Menges, F.D. Smith, and J.B. Smith. Architecture of the Artifact-Based Collaboration System Matrix. In *Proceedings of CSCW '92*, November 1992.

[2] Jin-Kun Lin. Virtual Screen: A Framework for Task Management. In *Proceedings of the Sixth Annual X Technical Conference*, January 1992.

[3] John Menges. The X Engine Library: A C++ Library for Constructing X Pseudo-Servers. In *Proceedings of the Seventh Annual X Technical Conference*, pages 129–141, 103 Morris Street, Sebastopol, CA 95472, January 1993. O'Reilly & Associates, Inc.

[4] D. Shackelford, J.B. Smith, and F.D. Smith. A Distributed Data-Storage Service for Supporting Group Collaborations. Technical Report TR92-044, Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina 27799, October 1992.

[5] J. B. Smith and F. D. Smith. ABC: A Hypermedia System for Artifact-Based Collaboration. In *Proceedings of Hypertext '91*, December 1991.

A Callbacks to C++ Code

Tcl and Tk have mechanisms for calling C procedures from the Tcl interpreter. These C procedures are called *callbacks*. For example, Tcl commands can be implemented in C, and the C callback routines implementing these commands can be *registered* with the Tcl interpreter using the **Tcl_CreateCommand** routine. Registering a command with the interpreter associates the callback routine with a command name. When the command is interpreted, the corresponding C callback routine is called.

The callback routine for command implementations has a fixed *signature* (**Tcl_CmdProc**) that defines the result and parameter types of the callback routine. Tcl variable reads and writes can also be *traced*, meaning that a designated C callback routine is called whenever the traced variable is read and/or written. The trace callback routine signature (**Tcl_VarTraceProc**) is different from the **Tcl_CmdProc** signature.

In C++ code, we can use the "extern C" declaration to define C++ routines with C calling conventions that can be used as callback routines called by the Tcl interpreter. For the C++ programmer, however, it is often desirable to have the interpreter call particular member functions of particular objects.

Tcl callback registration routines can be passed a **ClientData** pointer which can point to a structure of an arbitrary type. This **ClientData** pointer is then passed back as an argument to callback routines. The structure pointed to by the **ClientData** parameter can be used to store pointers to the the object and member function implementing a callback. Since we now have a C++ interface to the Tcl Interpeter, we can add our own registration functions to that interface which substitute the object and member function parameters for the C callback procedure pointer, as shown in Figure 3. Note that we have not replaced the original **Tcl_CreateCommand** interface; we have just used operator overloading to make a new one with different parameters. The Callback parameter specifies the object, and the CreateCommandCallback parameter specifies the member function of the Callback class that implements the command.

Our new CreateCommand registration routine (Figure 4) now makes its own **ClientData** structure to store the object and member function, along with the original **ClientData** pointer and a pointer to the Tcl Interpreter object. It then calls the original **CreateCommand**, registering the same **extern "C"** callback routine (**CallCmdProc**) for every command registration.

CallCmdProc (Figure 5) will be called whenever any command registered with our new CreateCommand procedure is interpreted. It is the responsibility of the CallCmdProc to actually call the appropriate member function of the appropriate object. It does so by using

```
class Tcl_Interpreter
{
public:
  typedef int
    (Callback::*CreateCommandCallback)
      (ClientData date,
       Tcl_Interpreter *interp,
       int argc,
       char *argv[]);

  void
  CreateCommand(const char *,
                Tcl_CmdProc *,
                ClientData,
                Tcl_CmdDeleteProc *)
  void
  CreateCommand(const char *,
                Callback *,
                CreateCommandCallback *,
                ClientData,
                Tcl_CmdDeleteProc *)
  ...
}
```

Figure 3: CreateCommand Interface

```
typedef struct {
  Tcl_Interpreter *interp;
  Callback *object;
  CreateCommandCallback *member;
  ClientData data;
} CmdProcData;

Tcl_Interpreter::CreateCommand
  (const char *cmdName,
   Callback *object,
   CreateCommandCallback *member,
   ClientData data,
   Tcl_CmdDeleteProc *deleteProc)
{
  CmdProcData* cpdata = new CmdProcData;

  cpdata->interp = this;
  cpdata->object = object;
  cpdata->member = member;
  cpdata->clientData = clientData;

  CreateCommand(cmdName,
                CallCmdProc,
                (ClientData) cpdata,
                deleteProc);
}
```

Figure 4: CreateCommand Implementation

```
extern ``C''
static int CallCmdProc
  (ClientData data,
   Tcl_Interpreter *interp,
   int argc,
   char **argv)
{
  CmdProcData* cpdata =
    (CmdProcData*) data;

  return
    ((cpdata->object)->*(cpdata->member))
      (cpdata->clientData,
       cpdata->interp,
       argc,
       argv);
}
```

Figure 5: CallCmdProc Implementation

the C++ member pointer selector operator.

The interpreter class, which makes it possible to define our own callback registration interfaces, and the **ClientData** parameters to Tcl callback registration routines thus make it possible to do what we want, i.e., to enable particular member functions of particular objects to be registered as callback routines. The above implementation, however, is not as flexible as we'd like.

To understand why, let's step back a bit and see exactly what we want the **CallCmdProc** procedure to do. We would like to make a call something like this:

```
return object->*member(...);
```

where **object** is an object of an arbitrary class and **member** is an arbitrary member of that class (with the proper signature). We accomplished the first objective by making **object** an instance of a subclass of the Callback class, using multiple inheritance as necessary. Unfortunately, **member** must be a member function of the Callback superclass, not of the dynamic class of **object**. (Some compilers will allow (albeit with warnings) a member function of the dynamic class of **object** to be assigned to a member pointer variable statically declared to refer to a member of the Callback superclass, but this is not safe because the compiler cannot ensure that the dynamic class of **object** has the function referred to by **member**.)

What this requires us to do is to create some fixed number of virtual member functions in class Callback which are re-implemented in each class that inherits from Callback. A pointer to the associated member function of the Callback superclass is then passed to **CreateCommand** as the **member** argument. This is not a nice solution, because we want to be able to have an arbitrary number of callback routines implemented

```
class Tcl_Interpreter
{
  ...
  void
  CreateCommand(const char *,
                Callback *,
                ClientData,
                Tcl_CmdDeleteProc *)
  ...
}
```

Figure 6: New CreateCommand Interface

by any given object (say, to implement multiple commands or traces on multiple variables). We also would like to be able to name our member functions in the derived class appropriately, instead of having to use the fixed names defined in the Callback superclass. It would also be nice if we didn't have to make every class that implements callbacks inherit from the Callback superclass.

Our solution is to introduce a level of indirection. We redefine **CreateCommand** to take a single **callback** object instead of an (object, member function) pair (Figures 6 and 7). This **callback** object inherits from a Callback superclass, and its job is to "know" the real object and member function that comprise the real callback routine and to perform the actual callback. The **CallCmdProc** procedure "calls" the callback (using the **operator()(...)**) operator (Figure 8). It no longer needs to know anything about the type of the member function that implements the callback.

The Callback and Callback_Objecttype classes are shown in Figures 9 and 10. The Callback_Objecttype constructor saves away the object and member function that implement the callback. The Callback_Objecttype implementation of the **operator()(...)** operator actually calls the callback routine.

Note that a separate class of the form of Callback_Objecttype is needed for each type of object that implements callbacks and for each callback signature type. Implementing a separate class for each object type can probably be eliminated by using C++ templates. We have not yet used templates because they are not universally available and we do not wish to introduce portability problems.

Now notice that it is not really necessary for the member function implementing the callback and the **operator()(...)** operator of the Callback class to have the same signature. This allows us to use the Callback_Objecttype class to convert the signature of the Tcl callback routine (**Tcl_CmdProc**) to something simpler. For example, we may create a Tcl command that always takes only one integer argument. The member function implementing the call-

```
typedef struct {
  Tcl_Interpreter *interp;
  Callback *callback;
  ClientData data;
} CmdProcData;

Tcl_Interpreter::CreateCommand
  (const char *cmdName,
   Callback *callback,
   ClientData data,
   Tcl_CmdDeleteProc *deleteProc)
{
  CmdProcData* cpdata = new CmdProcData;

  cpdata->interp = this;
  cpdata->callback = callback;
  cpdata->clientData = clientData;

  CreateCommand(cmdName,
                CallCmdProc,
                (ClientData) cpdata,
                deleteProc);
}
```

Figure 7: New CreateCommand Implementation

```
extern ‘‘C’’
static int CallCmdProc
  (ClientData data,
   Tcl_Interpreter *interp,
   int argc,
   char **argv)
{
  CmdProcData* cpdata =
    (CmdProcData*) data;

  return
    (*(cpdata->callback))
      (cpdata->clientData,
       cpdata->interp,
       argc,
       argv);
}
```

Figure 8: New CallCmdProc Implementation

```
class Callback
{
public:
  virtual operator()
    (ClientData data,
     Tcl_Interpreter *interp,
     int argc,
     char *argv[]) = 0;
}
```

Figure 9: Callback Class

```
class Callback_Objecttype
: public Callback
{
  typedef int
    (Callback_Objecttype::*
     CreateCommandCallback)
      (ClientData data,
       Tcl_Interpreter *interp,
       int argc,
       char *argv[]);

public:
  Callback_Objecttype(
    Objecttype *object,
    CreateCommandCallback member)
  : object(object), member(member) {};

  operator()
    (ClientData data,
     Tcl_Interpreter *interp,
     int argc,
     char *argv[])
  {
    return (object->*member)
      (data, interp, argc, argv);
  }
}
private:
  Objecttype *object;
  CreateCommandCallback *member;
}
```

Figure 10: Callback_Objecttype Class

back could just have one **int** argument, and the Callback_Objecttype class could perform the mapping between the **Tcl_CmdProc** signature and the simpler member function signature, checking to make sure the command was called with just one integer argument.