Cite as:
Roseman, M. and Greenberg, S. (1997). Building Groupware with GroupKit. In M. Harrison (Ed.) Tcl/Tk Tools, p535-564, O'Reilly Press.

15

Building Groupware with GroupKit
by Mark Roseman and Saul Greenberg

"Hmmm, that's interesting. Hey Linda, take a look at this."

"What do you have there Carl? Oh," Linda asked, glancing at the schematic on the screen.

"I want to get the signal here", Carl pointed, "over to this other part here, but I'm not sure the best way with all this stuff in between."

"Why don't you route it this way?" Linda suggested, drawing a rough path along the schematic, "this should be the most effective."

"Oh and then I can move this chip over to here instead," Carl replied as he dragged the chip to its new location.

Linda moved a couple of other components over. "That looks like it should pretty much do it."

"Great, that'll work. Thanks for your help."

They both turned back to their own work. A pretty typical situation that repeats itself thousands of times daily in workplaces everywhere.

Except that Linda and Carl are thousands of miles apart.

What is Groupware?

Linda and Carl are using a technology called groupware, which lets people who are far apart but connected by a network collaborate together on the same documents — such as the schematic in this example — at the same time. Although groupware had its beginnings with the visionary work of Douglas Engelbart in the 1960s, only recently have networks and workstations become powerful and ubiquitous enough to truly support it. Some examples of groupware systems in use today include shared electronic whiteboards, multi-user text editors, tools to support group brainstorming, and of course a wide variety of multi-player games.

Groupware actually refers to any technology that lets people work together. So things like email and Usenet news are also rudimentary groupware technologies. The difference is that email and Usenet don't allow people to work

together at the same time or "synchronously", but instead support different time or "asynchronous" work. Another distinction between types of groupware is whether the system supports people working in the same place (such as a team meeting room) or at a distance, such as different sites on the Internet. We're mostly concerned with groupware for geographically distributed groups working together synchronously.

Unfortunately, building groupware applications can be extremely difficult. Implementing even the simplest system is a lengthy and tedious process. Every application must worry about creating and managing socket connections, parsing and dispatching inter-process communication, locating other users on a network and connecting to them, keeping shared resources consistent between users, and so on. Using conventional programming tools, a lot of low level code must be written before getting to the specifics of the application. Groupware is an application domain crying out for better programming tools.

What is GroupKit?

GroupKit is an extension to Tcl/Tk that makes it easy to develop groupware applications to support real-time, distance-separated collaborative work between two or more people. Using Tcl's built-in socket commands for its low level networking, GroupKit provides an application framework that handles most details of building groupware automatically for you, so you can spend time just writing your application rather than its groupware infrastructure.

GroupKit grew out of our frustrations in building groupware systems without proper tools to support the job. By moving the common elements of groupware into a Tcl/Tk extension, we can now create programs in three days that originally took three months to write, and whose complexity shrunk from several thousand lines of code to only a few hundred lines. We've also found that it is often easy to take an existing single-user Tcl/Tk program and convert it to a multi-user program. With GroupKit, we're finding that writing groupware is only slightly harder than writing an equivalent single-user program.

The GroupKit distribution you'll find on the CD consists of the GroupKit extension itself (which is implemented as a mixture of Tcl and C), documentation, and approximately thirty example groupware applications.

Using GroupKit Applications — An End User's view

Before delving into the details of how to build GroupKit applications, we'll first walk through how an end user would run some of the existing applications included in the distribution. This will give you a chance to make sure things are set up on your own system, as well as introduce some important concepts about groupware programs.

We'll assume that GroupKit has already been compiled and installed on all participating systems, with a registrar process running. If GroupKit has not yet been installed, you can install it yourself by following the step by step installation instructions in the README file in the software distribution. In your own account, you'll need to make sure that your Unix PATH variable points to where the GroupKit binaries have been installed (type "which open.reg" at your shell prompt to check).

Starting a Session Manager

In GroupKit, you don't run programs directly, but invoke them via another program called a session manager. The session manager is used to locate other people in your work community running GroupKit programs, and connect your programs up to them, or to start up programs of your own. Running your programs together with other people is called a conference or session. GroupKit actually comes with several different session managers, but we'll use one called the Open Registration session manager, "open.reg". Start it by typing the following in a Unix shell window:

```
open.reg &
```

The first time you start up, the session manager may ask you some questions about yourself, such as your name, and store this in a file called ".groupkitrc" in your home directory. It should then show the window in Figure 1 (if you have problems, make sure that you have a registrar process running). The "Conferences" pane on the left shows the names of any running conferences. Selecting one will show who is in the conference in the "Participants" pane on the right. The "Conferences" menu contains a list of known GroupKit applications and lets you start up new groupware sessions.

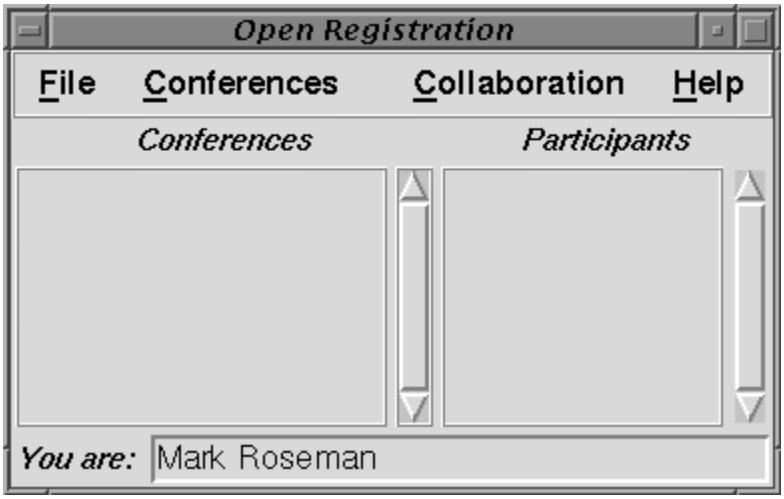

Figure 1. Open Registration session manager.

Creating a New Conference

Assuming there are no conferences already running which we could join, let's create a new conference. We'll use a program called "Simple Sketchpad", which acts like a shared whiteboard, allowing several users to simultaneously draw freehand on a canvas.

To create it, pull down the "Conferences" menu in the session manager and select "Simple Sketchpad". A dialog box will appear, shown in Figure 2. This dialog box allows you to give your conference a name (by default it is just the name of the application) which will identify it to other users. When you've picked a name, click the "Create" button.

Figure 2. Conference naming dialog.

At this point, you'll see the name of the conference added to the "Conferences" pane in the session manager, and the Simple Sketchpad program will come up in its own window, as shown in Figure 3. You can draw on the canvas using the left mouse button. The menus let you clear the canvas, exit the program, find out about other participants in the conference, and get information about both GroupKit and the Simple Sketchpad conference. Try it!

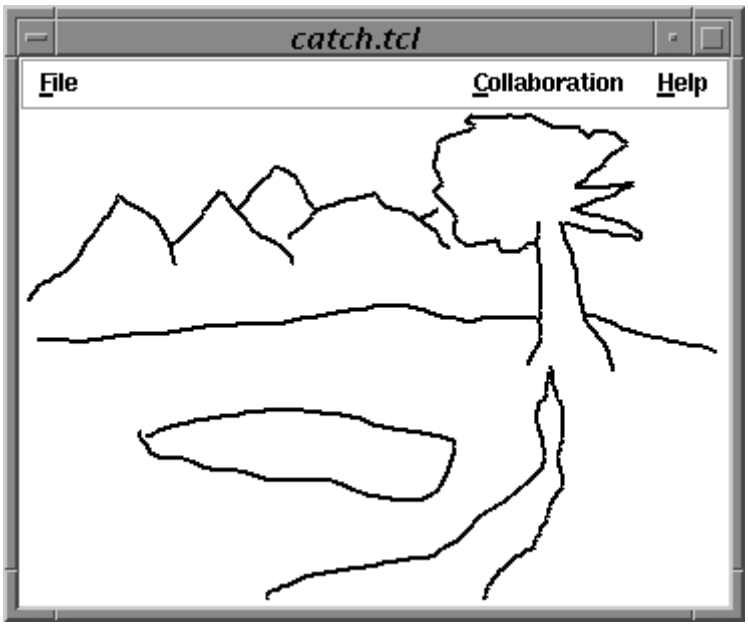

Figure 3. Simple Sketchpad conference.

Joining an Existing Conference

We'll now have another user join the conference you've created. Find someone else on another machine nearby, and get them to start up their own copy of the open.reg session manager. They should see the conference you've created, and clicking on it will show that you are a participant in it. If you are just trying GroupKit out by yourself, create another open.reg on the same machine, and pretend you are a different person by changing your name to a new one in the entry box on the bottom.

To join the conference from their session manager, the other participant can double-click the name of the conference. This will add their name to the list of participants, and also bring up a window with their own copy of the Simple Sketchpad program. You'll also see that any drawing that was done in the first copy of the program appears in the new copy.

You'll now find that both people can draw at the same time, and that any drawings made by one user immediately appear on the screens of the other user. You'll also see a small cursor called a telepointer, which tracks the location of the other user's mouse cursor as they move around the window. More than two people can be in this conference and others can join and participate through their own session managers.

When done, you can select "Quit" from the "File" menu of the Simple Sketchpad program to leave the conference. Your copy of the program will disappear, and you'll see your name removed from the list of conference participants in the session manager. When the last person leaves, they are asked to either save the contents of the conference (which gets stored in a background GroupKit process and would allow restarting at a later time, with the drawing intact), or to delete the conference.

Its also possible to invite other users into a GroupKit conference. In your session manager, select a conference, and then choose "Invite…" from the Collaboration menu. This will give you a list of all users who are currently running a session manager of their own. Choosing from that list will popup a message on their screen, asking if they would like to join the conference. If they decide to join, their session manager will automatically join them to the conference.

What's Really Happening Inside

To understand what is going on, lets take a step back. As you've noticed, GroupKit consists of a number of different processes, which are illustrated in Figure 4. There is a central process called the registrar, a typical Unix daemon that should already be running on your system. Its job is to keep track of what conferences are running and who is joined to them. Each user runs a session manager, such as open.reg, which connects up to the registrar. The session managers are used to create conferences (such as our Simple Sketchpad) which again run as separate processes. As other users join conferences through their own session managers, GroupKit opens up network connections between the conference processes, so that every process in a conference has a connection to every other process in a conference.

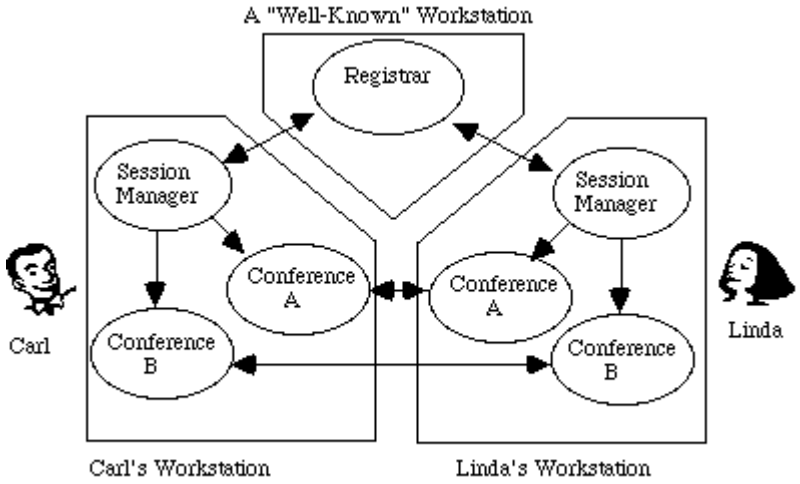

Figure 4. GroupKit process and communications architecture.

Other Applications

There are a number of other sample applications included with the GroupKit distribution which you should try out. Some of these are illustrated in Figure 5 and include:

Brainstorming Tool. Allows users to brainstorm by typing in brief one-line textual ideas; all ideas appear in a listbox visible by everyone.

File Viewer. Lets you load up the contents of a text file and browse through it. A multi-user scrollbar shows what parts of the file other people are looking at.

Hello World. A simple program that creates a button; when pressed, the button changes on all user's screens to say hello from whoever pushed the button.

Text Chat. A program similar to Unix talk, but allowing more than two participants. Text typed by each user is immediately seen by all others.

Post It. Type in a message, which can be sent to other users where it appears as a sticky note.

Tic Tac Toe. The classic game, allowing you to play against other users.

Tetrominoes. Rotate and move the different shaped polygons to get them to fit inside their containers.

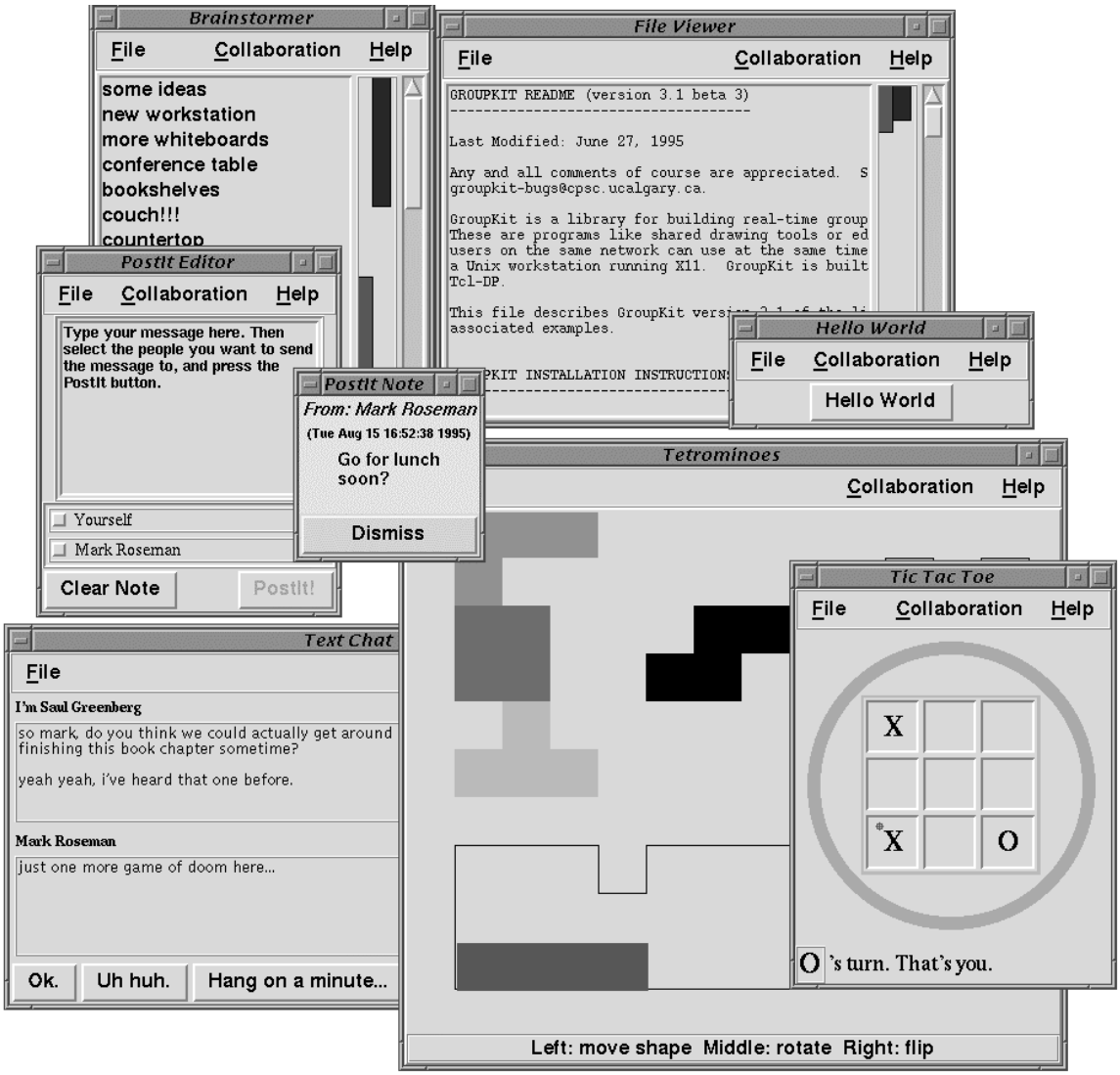

Figure 5. Some example GroupKit conference applications.

Important Concepts

There are several important points about GroupKit that have been illustrated so far.

- GroupKit supports real-time distributed multi-point conferences between many users.
- GroupKit systems include both session managers for managing conferences, such as the Open Registration system, and conference applications which are the actual groupware tools, such as Simple Sketchpad.
- Every user in a GroupKit conference session runs their own copy of the conference application in a process on their own machine. These processes are connected to each other over the network.
- GroupKit is not a media-space system, which would include things like audio or video conferencing. Many of the conference applications will strongly benefit from having some kind of voice connection, such as provided by a telephone. Alternatively, if you have some sort of computer-based media space system available, it would be possible to integrate it with GroupKit, so that starting a GroupKit conference also starts the media-space system.

An Example GroupKit Program

Now that we've seen what GroupKit programs look like and how its runtime architecture is set up, we can start building our own conference application. There are a lot of pieces in GroupKit to explain, so what we'll do is develop a relatively sophisticated groupware program over the space of the chapter, starting with a minimalist single-user version and gradually adding features that illustrate GroupKit's programming constructs.

The program we'll build is a brainstorming tool called "Note Organizer." This tool can help a group generate and organize ideas, for example to plan a paper, software project, advertising campaign, etc. As a brainstorming tool, the program will allow users to enter ideas (a few words), each which will be represented as a text item in a Tk canvas widget. To organize the ideas, users will be able to drag them around on the canvas, grouping related ones and so on. Being groupware, everyone in the group will be able to generate and move ideas around at the same time, and see what everyone else is doing. The program will look like the one in Figure 6.

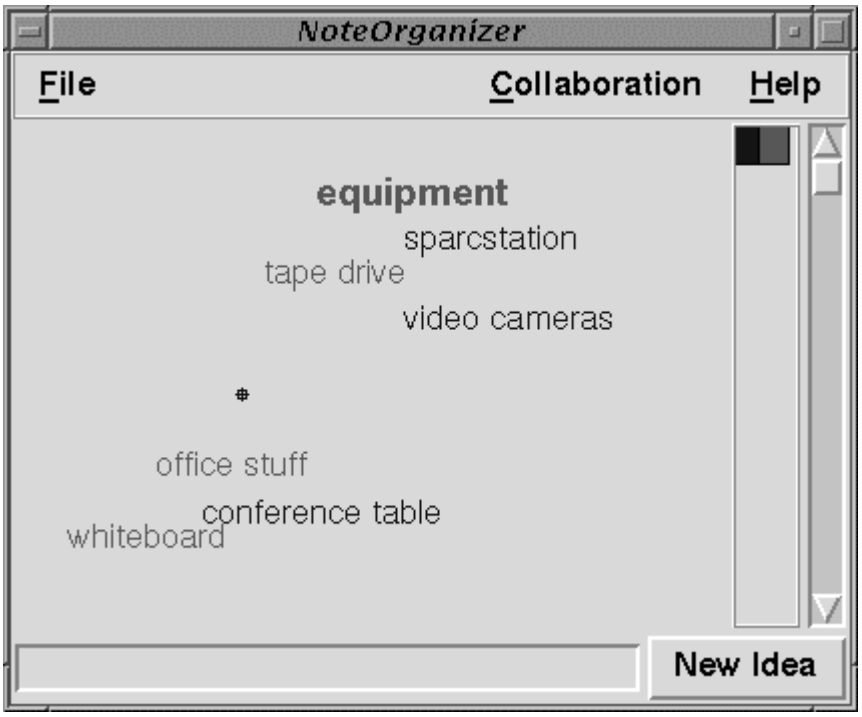

Figure 6. Screen dump of Note Organizer.

Here is how we'll approach building this example. We'll start off by creating a single-user version of the program that allows you to create ideas, but not to move them around. This will run as a normal "wish" script, without using GroupKit at all. We'll then take that program and modify it slightly so that it will run within GroupKit, before putting any code in to "share" the ideas. The next step will be to share ideas, so that when one participant creates an idea on their canvas, it appears on other participants' canvases also. We can then begin to get more sophisticated, by letting users drag ideas around the display, which will introduce some techniques that are useful when building groupware systems. Finally, we'll have the program do appropriate things when people enter and leave the session, and also add some groupware widgets that will help keep track of what people are doing when they are in the session with us. That will complete the application as illustrated in Figure 6. If you'd like to skip ahead and see what the entire program will look like, the complete listing of the final version appears at the end of this chapter.

Single-User Version

So lets start out with just a single-user version, and only worry about creating the ideas but not moving them around yet. The code — which is standard Tcl/Tk — is shown below. The program sets up some global variables to keep track of where the ideas are to be placed on the canvas, and the font to use when drawing the ideas. It then creates the canvas to hold the ideas, an entry widget to type in the ideas, and a button used to copy the idea from the entry to the canvas. When pressed, the button calls the `addNewIdea` proc, which gets the idea out of the entry, calls `doAddIdea` and clears the entry in preparation for the next idea. The `doAddIdea` procedure actually puts the idea into the canvas, by creating a text item at the location found in the `notes(x)` and `notes(y)` global variabless. It then adjusts these variables so as to place the next note below the one just added, starting a new column when it gets far enough down the canvas.

```
set notes(x) 20
set notes(y) 20
set notes(notefont) -adobe-helvetica-medium-r-normal--17-*

proc buildWindow {} {
    frame .main
    canvas .notepad -scrollregion "0 0 800 3000" -yscrollcommand ".scroll set"
    scrollbar .scroll -command ".notepad yview"
    frame .controls
    entry .newidea
    button .enteridea -text "New Idea" -command addNewIdea
    pack .main -side top -fill both -expand yes
    pack .notepad -side left -fill both -expand yes -in .main
    pack .scroll -side right -fill y -in .main
    pack .controls -side top -fill x
    pack .newidea -side left -fill x -expand yes -in .controls
    pack .enteridea -side left -in .controls
}

proc addNewIdea {} {
    set idea [.newidea get]
    doAddIdea $idea
    .newidea delete 0 end
}

proc doAddIdea {idea} {
    global notes
    .notepad create text $notes(x) $notes(y) -text $idea -anchor nw \
        -font $notes(notefont)
    incr notes(y) 20
    if {$notes(y)>600} {
        set notes(y) 20
        incr notes(x) 100
    }
}

buildWindow
```

You should be able to run that program just fine under Tk's normal wish. Type ideas in the entry box and press the button to add them to the canvas.

Running the Example in GroupKit

Now lets start turning this program into groupware. The first thing that every GroupKit program must do is initialize GroupKit, which is done by putting the following line at the beginning of your program:

```
gk_initConf $argv
```

The second thing you should do is add GroupKit's standard menubar, which contains menu items to exit the program, find out what other users are working on the program with you, and display an about box. Add these lines before creating the main interface of your program, i.e. just before creating the canvas widget:

```
gk_defaultMenu .menubar
pack .menubar -side top -fill x
```

As before, we'll start our conference application using the Open Registration session manager. But first we have to tell the session manager about our new program. To do this, add the following line to the bottom of your .groupkitrc file (which was created in your home directory the first time you ran the session manager). Of course, change the `/home/you` to the name of the directory where you put the note organizer script file of course.

```
userprefs prog.NoteOrganizer.cmd "exec gkwish -f /home/you/noteorg.tcl"
```

Either quit and restart your session manager, or choose "Re-initialize" from the "File" menu. You should now find an item named "NoteOrganizer" under the Conferences menu. Use it to create the Note Organizer conference as before, and then join the conference from another session manager.

If you run multiple copies, you'll quickly find that if you enter ideas in one copy of the program they don't appear on remote copies. Thats because we haven't actually told the program to display ideas on all screens — GroupKit won't take care of that automatically. You will find though that items in the menubar work fine, such as the "Show Participants" item which provides information on other users in the conference.

Just One More Thing...

So now lets fix our program so when ideas are entered in one copy of the program they are sent to other users. First, quit both running copies of the program (when you quit the last one it will ask you if you want to delete the conference or keep it around; you should delete it). Now, go back into your program and change the second line in addNewIdea from:

```
doAddIdea $idea
```

to:

```
gk_toAll doAddIdea $idea
```

and then re-start the program from the first session manager. After joining the conference from the second session manager, you should find that ideas entered in one copy of the program now appear in the other copy as well. That wasn't so bad!

So at this point, you've seen what is involved in getting a simple groupware program up and running in GroupKit. You've been exposed to GroupKit's session manager, learned how to tell the session manager about your new program, and how to initialize a GroupKit program. Finally, you've seen the `gk_toAll` command, which is one of GroupKit's programming constructs that can make it easy to turn a single-user program into a multi-user one.

So what does the `gk_toAll` do? That command arranges for the Tcl command following it (i.e. `doAddIdea $idea`) to be executed not only in the local program (where the idea was typed in), but also in every other copy of the program running in the conference. So, if you had not just two, but three, four or ten people joined in the conference, all of their conference processes would execute that same command. This is illustrated in Figure 7.

GroupKit Remote Procedure Calls

GroupKit's Remote Procedure Calls (RPCs) execute a Tcl command in the conference processes of different users in your conference session. They differ in which processes the commands are sent to.

gk_toAll *cmd args*. Execute a Tcl command on all processes in the session, including the local user.

gk_toOthers *cmd args*. Execute a Tcl command on all remote processes in the session, but not the process of the local user.

gk_toUsernum *user cmd args*. Execute a Tcl command on only a single conference process, which may be one of the remote users or the local process. The process is identified by its user's unique user number, which can be extracted from the `users` environment, described shortly.

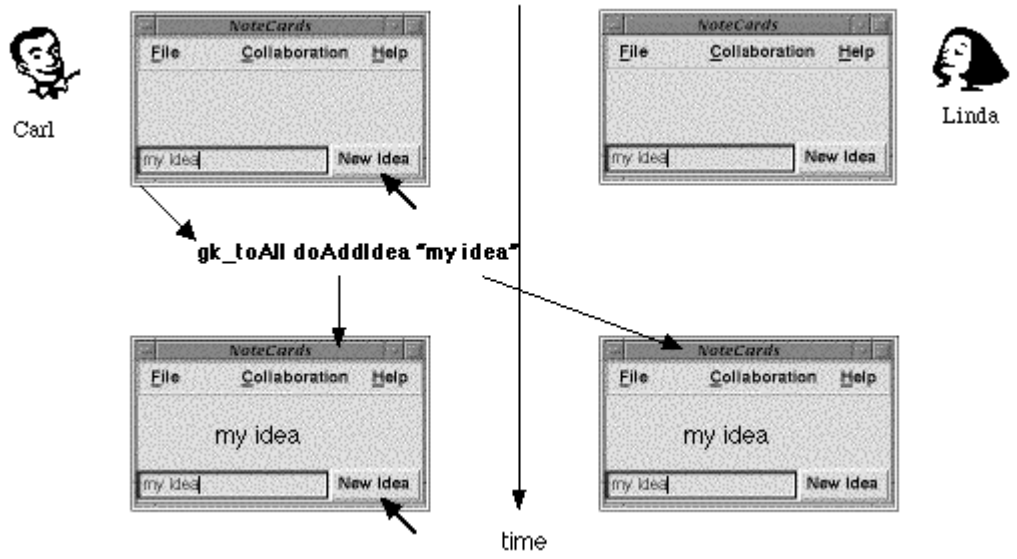

Figure 7. The gk_toAll command.

The `gk_toAll` command is an example of a Remote Procedure Call (RPC) that GroupKit provides. GroupKit has other RPC's that differ in who the commands get sent to, as shown in the sidebar. RPC's are a fairly straightforward but effective way to turn a single-user program into a multi-user one.

Coordinating Multiple Users

Now that our Note Organizer can create ideas that show up on everyones' screens, lets start looking at how we can move the ideas around to organize them. To accomplish this in a single-user program, we would change the doAddIdea procedure to attach a binding to each canvas item after it is created, and reposition the item when the mouse moved, as illustrated in the following code fragment.

```
set canvasid [.notepad create text $notes(x) $notes(y) -text $idea \
                  -anchor nw -font $notes(notefont)]
.notepad bind $canvasid <B1-Motion> "noteDragged $canvasid %x %y"

proc noteDragged {id x y} {
    .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
```

```
}
```

As before, we could use gk_toAll to execute that callback in all users' programs by changing the line in noteDragged to:

```
gk_toAll .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
```

This will work most of the time, but is very fragile and can break if users are entering ideas very quickly. Here's why. The code above assumes that copies of an idea on different users' screens all have the same canvasid. But that is not always true.

Tk assigns canvasid's in the order in which items are created. But if two users create ideas at almost the same time, they may end up being created in a different order on each users' screens. If the first user enters "foo", the gk_toAll adds the idea to the local canvas and then sends it across the network. If at the same time a second user enters "bar", their gk_toAll puts it on their canvas and sends it across the network. This will result in the first user adding "foo" then "bar", while the second user adds "bar" and then "foo." The two items will be created in different orders on the different systems, and therefore the canvasid's won't match. This will be a problem when it comes to moving the items around, because we need to correctly refer to the item to move. This is shown in Figure 8.

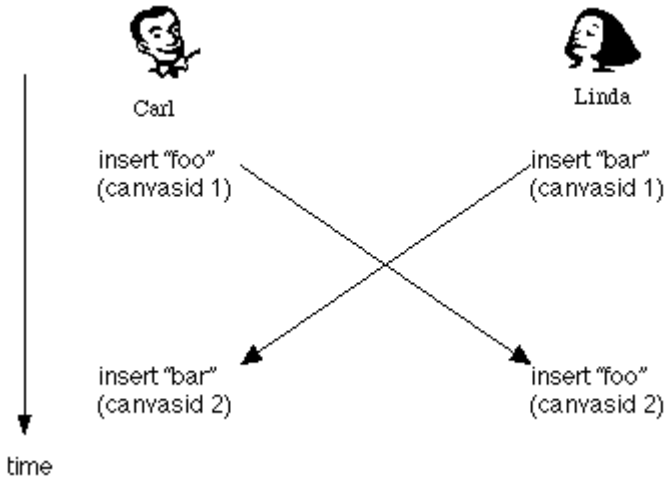

Figure 8. Ideas created in different orders on different machines.

To make matters worse, because the `doAddIdea` routine (which creates the text items on the canvas) also chooses the position on the canvas, if the ideas cross paths, not only will they have different canvasid's, but they will appear in different places on different users' screens.

Problems like these are fairly common in most groupware applications. They can be solved, but only if the programmer is aware of the nuances of these situations. .Lets address these problems one at a time.

Uniquely Referring to Ideas via Unique User Numbers.

The normal way you'd uniquely refer to an idea would be to give it a unique id number, such as via a counter held in a global variable that increments by one every time you have a new idea. As we saw with the canvasid's, that can create some problems in a groupware application, where the counter is not global across all processes.

What we'll do is slightly modify that scheme, so that each conference process will keep a global counter, but those counters will be unique from those of other conference processes. We will do this by prefacing each counter with a special GroupKit id number guaranteed unique for that conference process.

GroupKit generates a user number for every user in a conference process. It stores that number (and a lot of other information) in a data structure called an *environment*. We'll discuss environments in a moment, but for now just assume that the following code fragment will return the user's unique id number:

```
set myid [users get local.usernum]
```

We can now generate a unique id for every idea, by combining our user number (held in myid) with a global counter. We can then send that combined id to doAddIdea, and tag the canvas item with the combined id. Finally we can use this unique id when we're moving the idea around. This is shown below.

```
set notes(counter) 0

proc addNewIdea {} {
    global notes
    set idea [.newidea get]
    set myid [users get local.usernum]
    set id ${myid}x$notes(counter)
    incr notes(counter)
    gk_toAll doAddIdea $id $idea
    .newidea delete 0 end
}

proc doAddIdea {id idea} {
    global notes
    .notepad create text $notes(x) $notes(y) -text $idea -anchor nw -font \
        $notes(notefont) -tags $id
    .notepad bind $id <B1-Motion> "noteDragged $id %x %y"
     ...
}

proc noteDragged {id x y} {
    gk_toAll .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
}
```

Correctly Placing Ideas

So that takes care of uniquely referring to ideas. We will now fix the problem where ideas could end up being initially placed in different locations on different screens. This problem is really being caused because the receiver (doAddIdea) and not the sender (addNewIdea) is deciding where the idea should go. The code fragment below changes this so that the sender specifies the location, although we'll still let the receiver update the position for the next idea.

```
proc addNewIdea {} {
    ...
    global notes
    gk_toAll doAddIdea $id $notes(x) $notes(y) $idea
    ...
}

proc doAddIdea {id x y idea} {
    ...
    .notepad create text $x $y -text $idea -anchor nw \
        -font $notes(notefont) -tags $id
    ...
}
```

What happens now when ideas are entered at the same time by different users? If you trace through the order that messages will get executed, you'll see that both ideas will end up entered in exactly the same place, overlapping each other! When you think about it, this actually makes some sense. Imagine the real-life setting where users place real paper notes on a large board. If two people place notes at the same time, they'll both reach for the same place. Seeing their problem, they can then choose to move one of the notes somewhere else. In our computer version, users

can do the same thing — see the overlapping ideas and move one of them out of the way. While its probably possible to build very elaborate algorithms into software to recover from situations such as these, it is often best just to let people deal naturally with these situations.

Choosing what Information to Share

So far we've tried to keep the canvases of all users completely synchronized. This is known as a "What You See Is What I See" (WYSIWIS) view, where all participants see exactly the same thing. It is also possible to have some differences between displays, creating a "relaxed-WYSIWIS" view. GroupKit supports both paradigms, but choosing which information to share and which to keep private to a user is a design decision you have to make for your own application.

To illustrate relaxed-WYSIWIS, we'll add to our program the notion of a "selected" idea, just as you'd select text in an editor or an object in a drawing program. We'll display the selected idea in a larger font. For this application, we'll decide that each participant has their own selection. Users can select an idea by clicking on it. The code that actually manipulates the selection will keep track of the selected object by adding a "selected" tag to the canvas item. Here is the code:

```
set notes(selectedfont) -adobe-helvetica-bold-r-normal--20-*

proc doAddIdea {id x y idea} {
    ...
    .notepad bind $id <1> "selectNote $id"
    ...
}

proc selectNote id {
    global notes
    catch {.notepad itemconfig selected -font $notes(notefont)}
    catch {.notepad dtag selected selected}
    .notepad addtag selected withtag $id
    .notepad itemconfig $id -font $notes(selectedfont)
}
```

In this example, we made the conscious design decision to not share selection between users. However, it would of course be possible to have the selection shared, so that there is a single selection, shared among all users. This could be built by adding the appropriate gk_toAll RPC when changing selection.

We now successfully have dealt with the problems of coordinating multiple users. Generating unique id's based on the users' unique user number let us refer to objects uniquely, while letting the sender decide on the idea's position resulted in ideas appearing in the correct location on all screens. Finally, we showed that it may be desireable to have some information, such as selection, that is not shared between all users.

Using Environments to Find out Information about Users

At this point we can correctly create and move ideas, with the ideas appearing in the correct place on every user's screen. Lets now extend the program to identify who entered each idea. GroupKit keeps track of a designated color for each user which we can use to do this (participants can specify their personal color in their .groupkitrc file or via the "Pick Color" menu item in their session manager).

GroupKit stores each user's color (as well as other information) in a data structure called an environment. Environments are structured as a tree, where each node can contain either other nodes or a value. A node is referred to by its position in the tree, with each level of the tree separated by a dot. So for example, `local.usernum` refers to the node `usernum` which is a child of `local` which is a child of the root node of the environment. This is similar to how Tk refers to windows in the window hierarchy.

Using Environments

Environments have a number of different commands used to manipulate them. If you're familiar with Extended Tcl's keyed lists (see Chapter 6), you'll find some similarities. Some of the commands include:

1. gk_newenv *envname*. Create a new environment, which also creates a Tcl command called "envname" used to access the environment.
2. *envname* set *key value*. Set the value of the node located at key.
3. *envname* get *key*. Get the value of the node located at key.
4. *envname* keys *key*. List the children of the node located at key.
5. *envname* delete *key*. Delete the node located at key (and all nodes below it).
6. *envname* exists *key*. Check if a node located at key exists in the environment.
7. *envname* destroy. Destroy an environment, all data in it, and its Tcl command.

GroupKit maintains a number of environments internally. You've seen the `users` environment which holds information about all the users in the conference, including their color (the key "color"), their full name ("username"), their userid ("userid"), and host ("host"). You can also store your own information in the users environment, to be used by your application. The other environment you've encountered briefly is the `userprefs` environment, which is used within the .groupkitrc file to specify where different conferences exist. Environments are also used extensively throughout the session management subsystem in GroupKit.

As you've gathered, information about the local user is stored underneath the `local` key in the environment called `users`, so `local.color` is the node holding your own color. You can retrieve its value by the call `[users get local.color]`. Information about remote users is stored under a remote key, and further divided up by each remote user's unique user number, as shown in Figure 9. So for example, you can retrieve the color of the remote user having unique id 3 by specifying `users get remote.3.color`[*] . More examples of how to use environments are described in the sidebar "Using Environments".

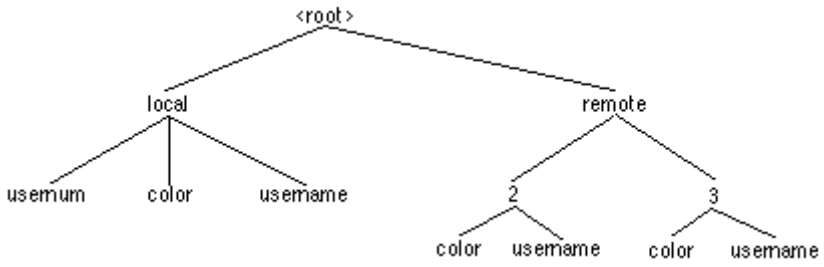

Figure 9. Users environment.

Lets add this into our program now. When we create an idea, we'll send our user number along with the idea. When creating the text item to display the idea, we'll change its color to be that of the user who created it. We'll also add a canvas tag to the text item identifying the user who created it, which we'll come back to later. Here are the changes in the code:

[*] For convenience, there is also a GroupKit call, "gk_getUserAttrib" which returns information about a user, and works for both the local user and remote users. So for example "gk_getUserAttrib $usernum color" would return the user's color.

```
proc addNewIdea {} {
    ...
    gk_toAll doAddIdea $id [users get local.usernum] $notes(x) $notes(y) $idea
    ...
}

proc doAddIdea {id usernum x y idea} {
    ...
    if {$usernum==[users get local.usernum]} {
        set color [users get local.color]
    } else {
        set color [users get remote.$usernum.color]
    }
    .notepad create text $x $y -text $idea -anchor nw -font  $notes(notefont) \
        -fill $color -tags [list $id user$usernum]
    ...
}
```

Environments provide a convenient way to find out information on other users. They also have other features that can help in building groupware, which we'll return to later. In the meantime however, we'll look at another GroupKit feature called conference events.

Conference Events

As you've noticed, GroupKit automatically takes care of users joining and leaving conferences, without your intervention. However, its often useful to be notified when people come and go. To do this, GroupKit provides a set of three conference events.

Conference events are similar to bindings that you can attach to widgets. But rather than executing a piece of callback code when something happens to a widget (e.g. the user presses a button), conference events execute a piece of code you provide as users join and leave the conference.

To specify handlers for conference events, you use the `gk_bind` command, which works very similarly to Tk's bind command, right down to its use of "percent substitutions" to pass event parameters to your callback code. The general format of the gk_bind command* is:

```
gk_bind event-type callback-code
```

New Users

The first of the conference events is the `newUserArrived` event, which signifies a new user has just joined the conference. We can extend our program to watch for new users arriving and displaying a dialog box with the code below. When this event is generated, information about the new user has already been stored in the `users` environment which was described above. One of the pieces of information available besides color is the user's name, so we'll display that in our dialog box. The percent substitution "%U" refers to the user's unique user number, a parameter of this conference event.

```
gk_bind newUserArrived {
    set newuser %U
    set name [users remote.$newuser.username]
    toplevel .new$newuser
    pack [label .new$newuser.lbl -text "$name just arrived."]
```

* While the gk_bind command creates a binding that will execute whenever a conference event occurs, it is also possible to later remove a binding, so that it will not execute when the event occurs in the future. The gk_bind command actually returns a "binding id", which can be later passed to the "gk_delbind" command to remove the binding.

```
    pack [button .new$newuser.ok -text Ok -command "destroy .new$newuser"]
}
```

Users Leaving

The second conference event is generated when a user leaves the conference. Again, the %U refers to the user's unique user number. The information for this user in the `users` environment is still available at the time this event is generated. In our program, we may choose to respond to this event by changing the color of any ideas generated by the leaving user to black, as illustrated in the code below (remember that we tagged all ideas created by that user with the word "user" appended with their user number).

```
gk_bind userDeleted {
    catch {
        .notepad itemconfig user%U -fill black
    }
}
```

Updating Latecomers and Saving Conference Results

The final conference event generated by GroupKit is the `updateEntrant` event, used to update latecomers to a conference already in progress. Unlike the previous two events, GroupKit sends this event to only one of the existing conference processes, chosen arbitrarily. The process receiving this event should respond to it by sending whatever information is necessary to the latecomer to bring them up to date. In our program, this would mean sending them a copy of all of the ideas that have been generated so far.

Figure 10. Updating latecomers.

This information is normally sent via the gk_toUserNum RPC, described earlier. This call needs the user number to send the message to, again extracted from the event via the "%U" substitution.

```
gk_bind updateEntrant {
    foreach item [.notepad find all]
        set x [lindex [.notepad coords $item] 0]
        set y [lindex [.notepad coords $item] 1]
        foreach tag [.notepad gettags $item] {
            if {[string range $tag 0 3]=="user"} {
                set usernum [string range $tag 4 end]
            } else {
                if {$tag!="selected"} {set id $tag}
            }
        }
        set idea [.notepad itemcget $item -text]
        gk_toUserNum %U doAddIdea $id $usernum $x $y $idea
    }
```

```
}
```

The updateEntrant event is also used to make conferences persist (that is, make them stick around after the last user has left, so that they can be rejoined later). You've noticed that when you quit the last program in the conference, you're asked if you'd like to delete the conference or have it save its contents (persist). If you ask it to persist, GroupKit sends an updateEntrant event to your program, but rather than asking it (via the %U parameter) to update a new user, GroupKit passes a special user number that causes your program to send messages to a special persistence server that records them. When a user next joins the conference, the messages stored in the server are played back, exactly as if they were sent from the last user.

It is also possible to create your own events, specific to your application. This can be useful as your programs get large, as a way of communicating changes between different parts of your program. See the gk_notifier(n) manual page for more information.

Groupware Widgets and Awareness

Our program is starting to get quite sophisticated, allowing ideas to be created and moved around on multiple screens, displaying different users' ideas in different colors, responding to new and leaving users, updating latecomers to the conference, and even persisting when all users have left the conference.

As a conference participant, you may sometimes find it hard to follow when several people are working at the same time. Ideas are being rapidly moved around, and its unclear who's doing what when. Also, because the canvas is quite tall, it can become hard to track where people are working.

GroupKit provides a number of different awareness widgets to help with these sorts of problems. Keeping track of where people are working and what they are doing is a common problem in a number of different groupware programs, so our widgets can be easily added to many different applications.

Telepointers

You've probably noticed yourself that when people get together around a whiteboard to discuss some sort of problem, that along with a lot of drawing, there's also a lot of pointing or gesturing to objects drawn on the whiteboard as well. In fact, some studies of how people use drawing surfaces like whiteboards found that gesturing is actually more common than drawing. GroupKit's telepointers provide a way to communicate this important gesturing information in groupware applications.

Telepointers, also known as multiple cursors, are used to provide very fine-grained information about where other users are working in the application. As other users move around the application, you'll see a small cursor which follows their mouse cursor. Just tracking those cursors — which you're usually just peripherally aware of — provides a surprising amount of information about who's doing what in an application, what objects people are working with, and how active different people are (Figure 11). Telepointers can be attached to any Tk widget.

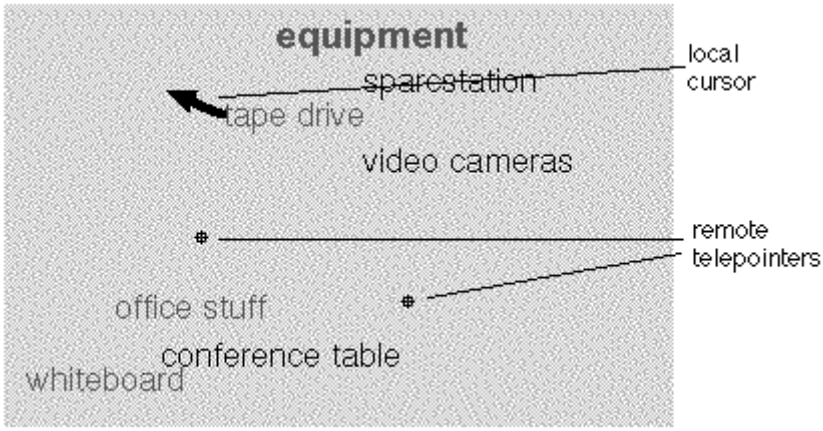

Figure 11. GroupKit's telepointers.

Adding telepointers to our example is quite straightforward, requiring only two lines of code. The first line initializes the telepointers, and the second attaches the telepointers to our canvas widget.

```
gk_initializeTelepointers
gk_specializeWidgetTreeTelepointer .notepad
```

Multi-user Scrollbars

While telepointers are good at answering questions like "what are people who are working close to me doing exactly?", multi-user scrollbars answer questions like "where are other people working in this large document?" They provide a more coarse-grained sense of awareness.

Figure 12 illustrates GroupKit's multi-user scrollbars. They consist of two parts: a conventional Tk scrollbar on the right, and a set of bars showing where users are located in a document to its left. The conventional scrollbar is used normally, to scroll your own view in the document. As you scroll, you'll also notice one of the bars to the left scrolling with you. That bar is showing your position in the document. As other users scroll their own views, the other bars on their display will change to show their new positions. Clicking on each bar will display a popup reminding you which user the bar is associated with. Multi-user scrollbars allow you to quickly find where others are working in a large document and — by aligning your view with theirs — join them.

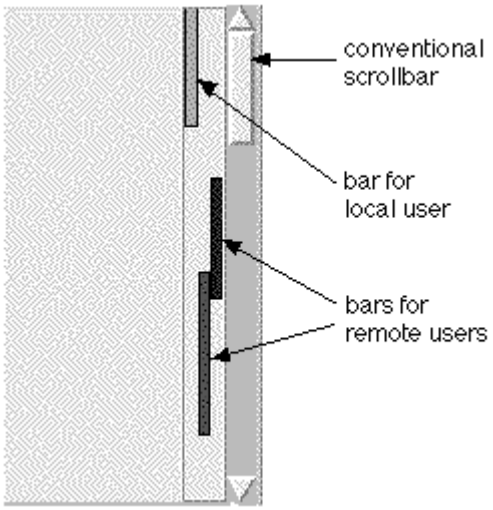

Figure 12. Multi-user scrollbars.

Multi-user scrollbars are added to applications in exactly the same way that conventional scrollbars are. In fact, the only difference is that instead of using the Tk `scrollbar` command, you use the GroupKit `gk_scrollbar` command.

```
proc buildWindow {} {
    ...
    gk_scrollbar .scroll -command ".notepad yview"
    ...
}
```

GroupKit also comes with a number of other widgets. You've already seen the menubar, which includes items such as an Exit command, an about box for GroupKit, and an item which brings up a dialog box giving you information about the different users in the conference with you. There's also a widget called the "mini view", which is similar to the multi-user scrollbar, but when combined with a text-based program can show a miniature of the document along with the scrollbar. There's also a widget that is included to make it easier to build online help into your application.

At this point we've completed our Note Organizer program; you'll find the complete program listing at the end of this chapter. The next section will go on to talk about a very different approach to building groupware applications with GroupKit.

Shared Environments

When writing the Note Organizer application, we've used a very direct style of programming; when we wanted to move an idea around, we literally sent a message to the other users' canvas widgets to move the idea. This approach works very well for small, highly graphical applications, but doesn't always scale up well to larger applications, particularly if you'd like to allow different users to customize how they work with their applications. This section discusses an alternative approach to programming in GroupKit, using some features of environments.

To understand how environments can be used to build groupware programs, we'll have to first describe a paradigm called "Model-View-Controller" (MVC), which was first introduced in Smalltalk as a way to construct (single-user) user interfaces. The MVC paradigm suggests that an application should be divided into three distinct components:

1. A "model" represents the underlying data structure that is to be displayed.
2. A "view" looks at the contents of the model and creates an on-screen visual representation of it.
3. A "controller" reacts to user input events and sends changes to the model.

Under this paradigm, the controller never changes the onscreen view directly, though the changes it makes to the model are picked up the view which does cause a change onscreen. This is illustrated in Figure 13.

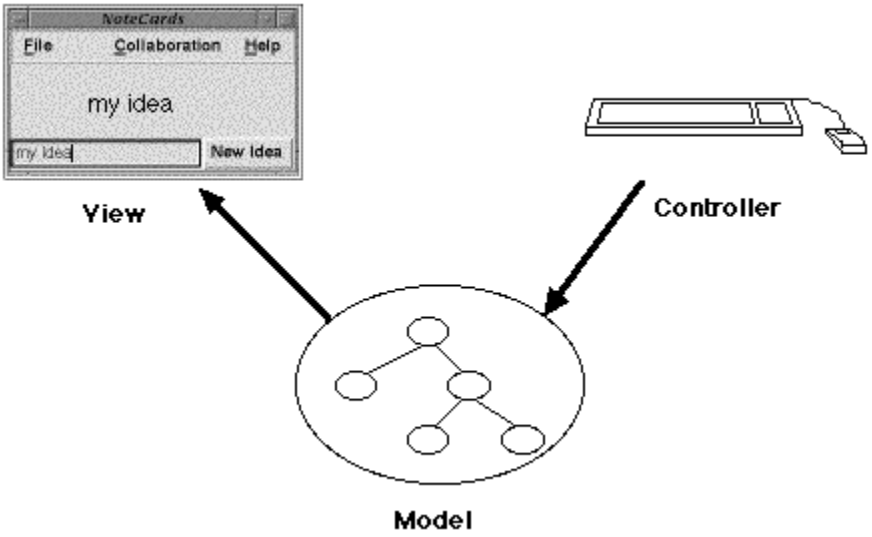

Figure 13. Model-View-Controller paradigm.

This approach can be a bit hard to follow at first, since input events only indirectly affect what is displayed onscreen, but has a number of advantages. First, by clearly separating out your underlying data from the view of that data and how it is manipulated, it encourages you to build a very clear data model, which can help you understand exactly what it is your program is manipulating. By doing this separation, you also find it easier to modularize your code, rather than trying to do everything in one place. Finally, the paradigm makes it easier to have multiple views on the same data structure, for example allowing you to view a table of numbers as either a spreadsheet or a graph.

So how does this relate back to GroupKit? Recall that we can use GroupKit's environments to store a fairly arbitrary set of data (such as names and colours in the `users` environment), arranged in a tree structure, where we can associate a string value with any node in the tree. This can serve as an underlying data structure for a wide variety of applications, and so is an appropriate choice for our Model under the MVC paradigm. Environments can also generate events (similar to the conference events used for announcing new users etc.) when they are changed. These events can be received by other portions of your program, such as a View which updates the screen to match the change in the Model. Finally, parts of your program that handle user input (the Controller portion) can react to these changes by modifying the environment.

We've taken environments one step further, and allowed them to be shared between all the users in a conference. When any user makes a change to their own copy of the environment, that change is automatically propagated to all other users' copies. So while a user's input event may change their own environment, which generates an event to update their onscreen view, the change *also* gets sent to every other user's environment, generating the same event and updating their view — instant groupware.

These shared environments and the MVC paradigm really shine on larger applications, but here we'll just present a toy example to illustrate the mechanisms. Our data model will be held in an environment called `stocks` and consist of a single piece of information, the value of the fictional GroupKit Inc. stock. Our view will consist of an onscreen label showing the value, and an entry widget will allow us to change the value.

After initializing GroupKit, we begin by creating the `stocks` environment, using the `gk_newenv` command. The `-share` flag tells GroupKit to create this as a shared environment* , and the `-bind` flag specifies that events should be generated when the environment is changed. We'll then give our GroupKit stock an initial value.

* Shared environments bring up the issue of concurrency control — what happens when two users do things at the same time? The default (specified with the "-share" flag) is to completely ignore this, which can lead to copies of the environments being out of sync. If you instead specify the "-serialize" flag, the copies will be guaranteed to be synchronized with each other, although

```
gk_initConf $argv
gk_defaultMenu .menubar
pack .menubar -side left -fill x
gk_newenv -share -bind stocks
stocks set groupkit 1
```

Next, we'll create our view, using a label widget. The `stocks bind changeEnvInfo` attaches a binding to the environment, so that whenever a change is made to it, the following code will be executed. There are also events for when a piece of information in the environment is first added (`addEnvInfo`), and when an item is removed (`deleteEnvInfo`)* . The event parameter %K refers to the name of the key in the environment that changed; we could have omitted they key here since there is only one key in the environment.

```
label .view -text "GroupKit Inc. value is [stocks get groupkit]"
pack .view -side top
stocks bind changeEnvInfo {
    if {%K=="groupkit"} {
        .view config -text "GroupKit Inc. value is [stocks get groupkit]"
    }
}
```

Finally, we create the controller, using an entry widget. When we hit the "Return" key in the entry, its value is retrieved and stored in the stocks environment. The view code should then notice the change and update the display. Finally, because it is a shared environment, the change will show up in all other users' environments, updating their views as well.

```
pack [entry .controller] -side top
bind .controller <Return> {stocks set groupkit [.controller get]}
```

We could easily replace either the controller or view with ones that behaved differently, without changing any other parts of the program (or they could even display several views, since multiple bindings on an event are allowed). Different users could also use different versions of the programs in the same conference. This works because only the models (the shared environment `stocks`) of different users directly communicate with each other, and not the views or controllers. So for example here is a view that uses a canvas widget to display a horizontal bar whose length depends on the value of the stock held in the environment:

```
pack [canvas .barview] -side top
.barview create rectangle 0 0 [expr [stocks get groupkit]*5] 10 -anchor nw -tags bar
model bind changeEnvInfo {
    if {%K=="groupkit"} {
        .barview coords bar 0 0 [expr [stocks get groupkit]*5] 10
    }
}
```

In summary, GroupKit's shared environments can serve as an easy way to implement the Model-View-Controller paradigm, and extend it to support multi-user applications. Environments can take care of the housekeeping chores in updating and synchronizing multiple copies of a data structure (such as the multiple copies of the canvas widget used in the Note Organizer — though the canvas data structure had the advantage of having its view built-in!), and can help to modularize your application. Its a judgement call whether to use shared environments or the more direct RPC's in your application (they can be mixed for different parts of course). In general, if your application becomes

---

this can cause some delays when running over slow networks. It is also possible to add other concurrency control policies into environments, though we're not going to get into that here.

* There is actually one other event generated by environments, the "envReceived" event. When a new user joins an already running session, GroupKit's environments will respond to an "updateEntrant" event by sending the entire contents of the environment to the new user. The "envReceived" event is generated at this point. The typical response by the programmer is to walk through the environment and create objects in your view to match what is already in the environment.

larger, if you're dealing with information that can be changed in various ways by your program, or if it can be viewed in different ways, then environments are usually an appropriate choice.

Other GroupKit Features

There's a lot more to GroupKit than we can cover here. In this section, we'll briefly mention a few more things, and also provide a set of pointers to more information.

Session Management

In this chapter, you've learned the basics of building a GroupKit conference application, such as the Note Organizer. To run it, you've used the open.reg session manager included with the GroupKit distribution. GroupKit actually comes with a number of different session managers, and also provides facilities for building your own, just as you can build your own conferences. Session management can and should be heavily influenced by the working style of a community of users (for example, should just anyone be able to join an existing conference?), and GroupKit makes it possible to meet the diverse needs of different groups.

Class Builder

GroupKit also comes with a mega-widget framework called the "GroupKit class builder", which allows you to build new types of widgets (and was used for those in GroupKit, such as the multi-user scrollbars). The class builder bears a strong resemblance to the Tix extension (see Chapter XX) and is in fact based on an earlier version of Tix.

High-Level Event Package

The gk_bind command used to send conference events is actually built on a more general high-level event package. This package can be used to broadcast arbitrary messages from one part of your application, which can be intercepted by other parts. Such high-level events are useful to help structure large programs, groupware or not. The gk_notifier object is the basis of these high-level events.

For more information

The GroupKit distribution contains a set of manual pages describing the various commands and widgets, and their options. You'll also find a user's manual (in Postscript) format. But as usual, the best way to learn GroupKit is to look at other people's programs. The distribution contains approximately 30 example conferences and session managers, written by ourselves and also contributed by other users of GroupKit. These examples are a great starting point for your own programs.

The most up-to-date information about GroupKit can be found on our World Wide Web page, accessible at http://www.cpsc.ucalgary.ca/projects/grouplab/groupkit. There you'll find information on current projects, and pointers to various papers that have been written about GroupKit. The latest version of the software can be found on our ftp site, ftp.cpsc.ucalgary.ca, in the directory /pub/projects/grouplab/software. Finally, we run a mailing list, which you can join by sending email to groupkit-users-request@cpsc.ucalgary.ca. The sidebar "Further Reading" describes a number of sources for more information on groupware in general.

Further Reading

There are a number of excellent souces of information about groupware, as well as Computer Supported Cooperative Work (CSCW), which is the academic discipline that encompasses groupware. A recent collection is "Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration", written and edited by Ron Baecker and published by Morgan-Kaufmann in 1993. It includes not only technical groupware information, but also case studies of cooperative work, information on the psychology of groups, and other foundational areas whose understanding is critical to building successful groupware. Other good collections include "Computer-Supported Cooperative Work and Groupware" (edited by Saul Greenberg; published by Academic Press in 1991) and "Intellectual Teamwork: Social and Technical Foundations of Cooperative Work" (edited by Jolene Galegher, Rob Kraut, and Carmen Egido, published by Lawrence Erlbaum Associates in 1990).

The premier academic conference is the CSCW conference, sponsored by the ACM, which has been held every second year since 1986. Besides the conventional proceedings (available from ACM Press), recent years (since 1992) have also included a video proceedings. The European CSCW conferences have been held in odd years since 1989. Other conferences having a large concentration of groupware or CSCW papers include the annual ACM SIGCHI conference (dealing with user interface and human factors issues), as well as the ACM Organizational Computing Systems conference (formerly Office Information Systems). Communications of the ACM has also run special issues devoted to CSCW (December 1991, January 1993, January 1994). Dedicated journals include Computer Supported Cooperative Work (Kluwer) and Collaborative Computing (Chapman and Hall), though groupware articles can also be found in Transactions on CHI (ACM) and Transactions on Information Systems (ACM).

Finally, a couple of pointers on the Internet. Yahoo maintains a page of groupware information you can access via "Computers and Internet:Software:Groupware". Pal Malm has collected together a number of systems and projects into the "unofficial CSCW Yellow Pages". Another good page of pointers is maintained by Tom Brinck.

- http://www.yahoo.com
- ftp://gorgon.tft.tele.no/pub/groupware/cscw_yp.ps.Z
- http://www.crew.umich.edu/~brinck/cscw.html

Complete Program Listing

Below is the complete code for the Note Organizer program that was built up during this chapter.

```
gk_initConf $argv
pack [gk_defaultMenu .menubar] -side top -fill x

set notes(x) 20
set notes(y) 20
set notes(counter) 0
set notes(notefont) -adobe-helvetica-medium-r-normal--17-*
set notes(selectedfont) -adobe-helvetica-bold-r-normal--20-*

proc buildWindow {} {
    frame .main
    canvas .notepad -scrollregion "0 0 800 3000" -yscrollcommand ".scroll set"
    gk_scrollbar .scroll -command ".notepad yview"
    frame .controls
```

```
    entry .newidea
    button .enteridea -text "New Idea" -command addNewIdea
    pack .main -side top -fill both -expand yes
    pack .notepad -side left -fill both -expand yes -in .main
    pack .scroll -side right -fill y -in .main
    pack .controls -side top -fill x
    pack .newidea -side left -fill x -expand yes -in .controls
    pack .enteridea -side left -in .controls
    gk_initializeTelepointers
    gk_specializeWidgetTreeTelepointer .notepad
}

proc addNewIdea {} {
    global notes
    set idea [.newidea get]
    set myid [users get local.usernum]
    set id ${myid}x$notes(counter)
    incr notes(counter)
    gk_toAll doAddIdea $id [users get local.usernum] $notes(x) $notes(y) $idea
    .newidea delete 0 end
}

proc doAddIdea {id usernum x y idea} {
    global notes
    if {$usernum==[users get local.usernum]} {
        set color [users get local.color]
    } else {
        set color [users get remote.$usernum.color]
    }
    if {$color==""} {
        set color black
    }
    .notepad create text $x $y -text $idea -anchor nw -font $notes(notefont) \
            -fill $color -tags [list $id user$usernum]
    .notepad bind $id <B1-Motion> "noteDragged $id %x %y"
    .notepad bind $id <1> "selectNote $id"
    incr notes(y) 20
    if {$notes(y)>600} {
        set notes(y) 20
        incr notes(x) 100
    }
}

proc selectNote id {
    global notes
    catch {.notepad itemconfig selected -font $notes(notefont)}
    catch {.notepad dtag selected selected}
    .notepad addtag selected withtag $id
    .notepad itemconfig $id -font $notes(selectedfont)
}

proc noteDragged {id x y} {
    gk_toAll .notepad coords $id [.notepad canvasx $x] [.notepad canvasy $y]
}

gk_bind newUserArrived {
    set newuser %U
    set name [users remote.$newuser.username]
    toplevel .new$newuser
    pack [label .new$newuser.lbl -text "$name just arrived."]
    pack [button .new$newuser.ok -text Ok -command "destroy .new$newuser"]
}

gk_bind userDeleted {
    catch {
        .notepad itemconfig user%U -fill black
    }
}

gk_bind updateEntrant {
    foreach item [.notepad find all] {
        set x [lindex [.notepad coords $item] 0]
```

```
        set y [lindex [.notepad coords $item] 1]
        foreach tag [.notepad gettags $item] {
            if {[string range $tag 0 3]=="user"} {
                set usernum [string range $tag 4 end]
            } else {
                if {$tag!="selected"} {set id $tag}
            }
        }
        set idea [.notepad itemcget $item -text]
        gk_toUserNum %U doAddIdea $id $usernum $x $y $idea
    }
}

buildWindow
```