GROUPKIT USER'S GUIDE

Mark Roseman
Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4
Phone: +1 403 220-7691
E-mail: roseman@cpsc.ucalgary.ca

Introduction

GroupKit overview

GroupKit is a toolkit for developing real-time groupware applications. It consists of a set of C++ classes and applications, providing a number of common groupware components. GroupKit programs run on Unix machines, under X Windows. GroupKit is based on the InterViews user interface toolkit, developed at Stanford University.

About this Tutorial

This tutorial provides, by way of several examples, an introduction into developing groupware applications using GroupKit. By the end of this tutorial, it should be possible to develop and run simple groupware applications, using many of the important concepts and components provided by the toolkit.

Previous Knowledge

This tutorial (and the remainder of the GroupKit documentation) assumes a familiarity with C++ and InterViews, as well as some knowledge about groupware systems and their features. The tutorial also assumes that a C++ compiler, InterViews, and GroupKit have been installed on the development system.

Remainder of this Document

The tutorial consists of a number of parts:

1. A high-level view of the GroupKit application architecture.
2. Designing, implementing, compiling, and running a simple program under the GroupKit framework.
3. Adding messaging features to programs.
4. Using attribute lists to simplify sending complex messages.
5. Using overlays to provide common groupware features for work surfaces.
6. Monitoring when users enter and leave a conference.

For Further Information

The "README" file in the GroupKit distribution provides details for installing GroupKit on the development system.

The "GroupKit Reference Manual" provides detailed descriptions of the various C++ classes which make up GroupKit.

The paper "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications" (Proc. CSCW '92, Toronto, Ontario) describes the motivation for GroupKit's design as well as its implementation.

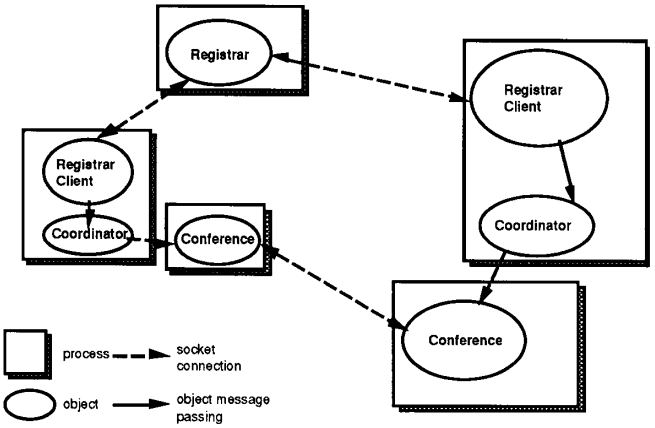
The "InterViews Reference Manual" (from the InterViews distribution, available from Stanford University) provides details of the InterViews class library.

The document "A Not-Entirely Gentle Introduction to InterViews" (included in this distribution) provides a brief introduction to InterViews.

GroupKit Architecture

Overview

GroupKit applications consist of a number of processes, arranged in a distributed architecture on a number of machines:



Registrar

The central Registrar maintains a list of all the conferences active on the system, including their users. The Registrar itself implements no policy on how conferences are created or deleted, or how users join or leave them, but simply maintains these lists.

Registrar Client

The Registrar Client (one per user) allows users to create, delete, join or leave conferences. It interacts with other Registrar Clients through the central list provided by the Registrar. The Registrar Client provides both a user interface as well as a policy dictating how conferences are created or deleted and how users enter or leave conferences. Different Registrar Clients can be created to suit different registration needs.

Coordinator

The Coordinator (one per user) acts as an interface between application conferences and the registration system. The Coordinator creates and maintains connections to any number of conferences, allowing them all to share the same registration mechanism.

Conference

The Conference is the heart of the groupware application itself, separate from the registration system. Conference applications maintain communication facilities necessary for exchanging messages with other Conferences. The user interface portions of groupware applications use these facilities extensively.

A Simple Example

About the Example

This section will start to build a GroupKit program. The example will be a multi-user version of the "patch" example contained in the InterViews distribution. The program uses an InterViews "deck" object to display one of three strings (cards) contained in the deck. Buttons for "previous" and "next" allow switching between the cards.

The multi-user version will broadcast changes to all users, so that when one user moves to a different card, other users will see the same card. The communications routines are detailed in the next section; this section describes how to get the basic single-user program up and running in the GroupKit framework.

Designing the ConferenceGlyph

The first step is designing the InterViews glyph which will manage the user interface. GroupKit programs subclass glyphs from the class ConferenceGlyph, which is an InterViews InputHandler containing a pointer to a Conference object. The Conference object allows the glyph to communicate with other conference users.

The glyph will contain the deck, a patch to redraw the deck when it changes, and buttons for previous and next. Callback routines for the buttons will be necessary, as well as a function "flip" which will be used by the callbacks. The declaration looks like this:

```
class FlipGlyph : public ConferenceGlyph {
public:
    FlipGlyph( WidgetKit*, Conference* );
private:
    void prev();
    void next();
    void flip(GlyphIndex cur);
    Patch* patch_;
    Deck* deck_;
};
```

The glyph will need to define callbacks for the buttons, which must be declared as in normal InterViews:

```
declareActionCallback(FlipGlyph)
implementActionCallback(FlipGlyph)
```

The constructor simply creates all the necessary components:

```
FlipGlyph::FlipGlyph(WidgetKit* kit, Conference* conf) :
        ConferenceGlyph(nil, kit->style(), conf)
{
  const LayoutKit& layout = *LayoutKit::instance();

  deck_ = layout.deck(
    kit.label("Hi mom!"),
    kit.label("Big Bird"),
    kit.label("Oscar")
  );
  patch_ = new Patch(deck_);

  body(
    kit.inset_frame(
      layout.margin(
        layout.vbox(
          patch_,
          layout.vglue(5.0),
          layout.hbox(
            kit.push_button(
                   "Next",
                    new ActionCallback(FlipGlyph)
                            (this, &FlipGlyph::next)),
            layout.hglue(10.0),
            kit.push_button(
                    "Previous",
                     new ActionCallback(FlipGlyph)
                            (this, &FlipGlyph::prev))
          ),
        ),
        10.0
      )
    )
  );
}
```

Next, the callback functions, "next" and "prev" are defined, along with the routine "flip" which actually alters the deck:

```
void FlipGlyph::prev() {
  GlyphIndex cur = deck_->card() - 1;
  flip( cur == -1 ? deck_->count() - 1 : cur );
}

void FlipGlyph::next() {
  GlyphIndex cur = deck_->card() + 1;
  flip ( cur == deck_->count() ? 0 : cur );
}

void FlipGlyph::flip(GlyphIndex cur) {
  deck_->flip_to(cur);
  patch_->redraw();
}
```

The Main Program

The main program is similar to main programs in InterViews. The one change is that rather than instantiating a Session object, a "GroupSession" is instantiated. A GroupSession defines several GroupKit style attributes, as well as parsing command line arguments received by all GroupKit programs from the registration system. The GroupSession object uses this information to instantiate a Conference object, which is needed by the ConferenceGlyph subclass defined previously.

```
int main(int argc, char** argv) {
  GroupSession* session =
              new GroupSession("DeckFlip", argc, argv);
  session->run_window(
    new ApplicationWindow(
      new FlipGlyph(WidgetKit::instance(),
                    session->conference())
    )
  );
}
```

Finally, some include files and callback declarations are needed at the beginning of the program:

```
#include <IV-look/kit.h>
#include <InterViews/layout.h>
#include <InterViews/style.h>
#include <InterViews/action.h>
#include <InterViews/patch.h>
#include <InterViews/window.h>

#include <gk/conference.h>
#include <gk/groupsession.h>
#include <gk/confglyph.h>
```

Compiling the Program

GroupKit uses Imakefiles to compile programs. Assuming the above program was stored in "deckeg.c", the following Imakefile would be used to compile it:

```
#ifdef InObjectCodeDir

Use_libInterViews()
MakeObjectFromSrc(deckeg)
GroupKitProgram(deckeg, deckeg.o)

#else

MakeInObjectCodeDir()

#endif
```

A script called "gkmkmf" should be installed, which works like the InterViews "ivmkmf" but also includes the necessary paths and Imakefile macros to easily build GroupKit programs. To compile the example use the following commands:

```
% gkmkmf
% make Makefiles
% make depend
% make
```

Informing the Registration System

GroupKit conference programs are not run directly by the user, but instead started by the registration system, which informs the conference program (through a number of command line arguments) about various things it needs to run, such as the location of the registration system (so that the conference can connect to the registration system and receive information about other users).

The registration system needs to know two things about each program that it can start. The first is the name (including path) of the executable programs which run the conference applications. The second is a brief description of the conference program, suitable for displaying to the user. These are specified through the standard X resources mechanism (i.e. .Xdefaults file). Assuming the deck flip executable was renamed to "deckeg" and placed in the "/home/grouplab/bin/SUN4" directory, the following lines could be added to the .Xdefaults file:

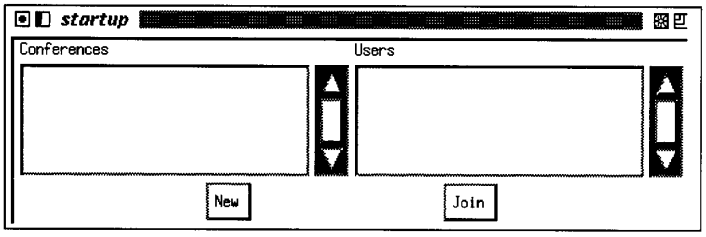
```
startup*GroupKitBinDir:      /home/grouplab/bin/SUN4
startup*conferenceTypes:     1
startup*conf1-desc:          Deck Flip Example
startup*conf1-prog:          deckeg
```

Here, "startup" is the name of the user registration program. Any number of conferences can be specified, by altering the "conferenceTypes" resource. Each program must have both a "conf<n>-desc" resource and also a "conf<n>-prog" resource.

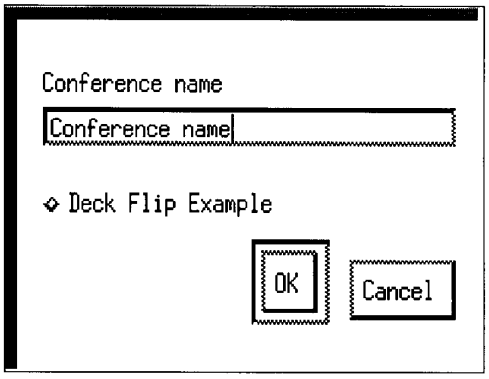
Running the Program

To run the program, first of all make sure the GroupKit registrar is running. If it is not running, start it up by typing "registrar &". Note that the registrar must run on a certain machine (specified by the "RegistrarHost" resource in the file "core-src/groupsession.c").

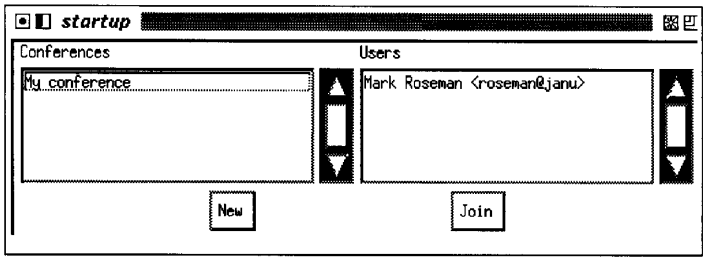
Next, run the program "startup", which provides a client to the registrar. This program expects to find the registrar at the host and port number identified by the RegistrarHost and RegistrarPort resources, but this can be overridden with the "host=<name>" and "port=<number>" command line options. This should bring up the window shown below. Note the appearance may be slightly different, depending on the default InterViews WidgetKit used (the screen dumps here show the monochrome WidgetKit).



Clicking the "New" button should bring up the dialog box below, asking you to create a new conference. The name of the conference can be entered in the field editor at the top, and the radio button for "Deck Flip Example" should be selected. If more conferences were specified in the .Xdefaults, more radio buttons would be in the dialog.



Clicking OK should start up the Deck Flip program in another window. The Conference list in the registrar client will be updated to include the newly created conference. Selecting the conference name will give a list of its users in the other list.



Other users may join the conference by selecting the conference name and pressing the "Join" button. This will bring up copies of the Deck Flip program on their screens, but right now the programs are behaving independently; the next section describes the routines necessary to communicate between the programs.

Using Messages

GroupKit Communications

GroupKit uses messages to communicate between different applications. A message consists of a "message-type" which identifies the message, and an "option-string" which provides any information needed by the message-type. Message-types are just integers; a number are predefined in "core-hdr/groupkit.h", but others can be trivially added. Option-strings are just normal strings, in any format (provided the message sender and message receiver use the same format!).

The remainder of this section will illustrate the use of messages, by adding them to the Deck Flip example above.

Defining Message Types

The first step is defining the message-types used by the application. In the Deck Flip case, only one message-type is needed, which will be used to tell other users to flip to a particular card in the deck. The option-string will consist of just a single integer (coded into a string) specifying the card to flip to. The message-type (any integer not already used) can be defined simply as follows:

```
#define DECKFLIPMSG   200
```

Sending Messages

In order to send messages, we'll use the ConnectionList in the Conference object, which holds Connections to all other users in the conference. The ConnectionList provides routines for sending messages to a single user (specified by their user number in the conference) or to all the users of the conference, which is necessary here.

The sending routines take a "message sender" object, which is capable of sending the same message on a number of Connections. We can create a message sender using the GroupKit class StrMsgSender, which takes an integer (the message-type), and a string (the option-string) as parameters in its constructor. The code to send a message would be placed in our flip routine as follows:

```
void FlipGlyph::flip(GlyphIndex cur) {
  char s[10];
  deck_->flip_to(cur);
  patch_->redraw();
  sprintf(s, "%d", cur);
  conference()->connections()->toAll(
              new StrMsgSender( DECKFLIPMSG, s) );
}
```

Receiving Messages

Two steps are necessary to receive messages. First, callback routines need to be specified, which will be executed when particular message types are received. In the example, a routine (here called "remoteFlip") should be activated when the DECKFLIPMSG is received. The following code in the FlipGlyph constructor specifies this:

```
conf->connections()->callbacks()->insert(
   DECKFLIPMSG,
   new StrActionCallback(FlipGlyph)
            (this, &FlipGlyph::remoteFlip));
```

Here, the ConnectionList in the Conference object maintains a table of callback routines used by every Connection in the list. The insert routine adds another callback to the table. StrActionCallback is a callback class specifying a routine taking a single string parameter (used here to specify the option-string). As with other InterViews callbacks, the StrActionCallback must be previously declared:

```
declareStrActionCallback(FlipGlyph)
implementStrActionCallback(FlipGlyph)
```

Finally, the target of the callback must be provided. This routine takes the parameters (in this case, the single integer for the card index) from the string and acts upon it. The action in this case will be to update the deck, as was done when the "Next" or "Previous" buttons were pushed. This is a common circumstance, having the same routine invoked locally (through the user interface) and remotely (by a message). It is useful to rearrange the code to provide one routine (here "real_flip") which performs the action, callable by two other routines. One of these routines ("flip") responds to local actions, and the other ("remoteFlip") responds to remote actions. This is illustrated in the code below:

```
void FlipGlyph::real_flip(GlyphIndex cur) {
  deck_->flip_to(cur);
  patch_->redraw();
}

void FlipGlyph::flip(GlyphIndex cur) {
  char s[10];
  real_flip(cur);
  sprintf(s, "%d", cur);
  conference()->connections()->toAll(
        new StrMsgSender( DECKFLIPMSG, s) );
}

void FlipGlyph::remoteFlip(char* s) {
  int which;
  sscanf(s, "%d", &which);
  real_flip(which);
}
```

Several more include files are now necessary at the beginning of the program:

```
#include <gk/infoconn.h>
#include <gk/straction.h>
#include <gk/msgsender.h>
#include <gk/reader.h>
```

The program will now provide a multi-user deck flip, where changes made by one user will be mirrored on other users' programs. A complete listing of the program can be found in Appendix A.

Using Attribute Lists

What Attribute Lists Provide

The previous section used sprintf() and sscanf() to encode and decode option-strings for transmission. This works well for simple situations. However, often a large number of parameters must be transmitted in the option-string. They are usually separated by some character, such as a colon. However, problems can arise when some of the parameters may be string, potentially containing colons, or during development, changes in the order or number of parameters must be carefully reflected in both sending and receiving routines.

Attribute Lists provide a general list of <name,value> pairs, where both attribute names and values are specified as strings. The information in these lists can be written onto a string for transmission, and a received string can be reconstructed into an Attribute List. The order of parameters in these strings is not important (although the names of the attributes must be the same on both sides) and separator characters (such as the colon) are escaped (with a "\") when necessary for transmission.

Making an Attribute List

Attributes are added to an Attribute List using the "attribute" method. Any attributes with the same name are replaced. The "remove_attribute" routine removes an attribute, while the "find_attribute" routine looks up an attribute's value. This is shown in the following example:

```
#include <gk/attrlist.h>
#include <stdio.h>

main() {
  char p[80], q[80];
  AttributeList* al = new AttributeList();

  al->attribute( "id", "123" );
  al->attribute( "name", "Mark" );
  al->attribute( "temp", "blah" );
  al->attribute( "id", "45" );
  al->remove_attribute( "temp" );

  if (al->find_attribute("id", p)) printf("%s\n", p);
  if (al->find_attribute("name", q)) printf("%s\n", q);
}
```

Transmitting an Attribute List

An AttributeList can be translated into a single string, suitable for transmission, using the write method. This could then be sent using a message sender, as done previously.

```
char s[1000];
al->write(s);
conference()->connections()->toAll(
        new StrMsgSender( WHATEVERMSG, s) );
```

Reconstructing an Attribute List

On the receiving end, the string could be reconstructed into an Attribute List using the static method read.

```
// string coming into our routine as "s"
char id[80];
AttributeList* al = AttributeList::read(s);
if (al->find_attribute("id", id))
   ...
```

Using Overlays

About Overlays

Human factors work in groupware systems has suggested that there are several activities which are common to a number of different types of conferences. These "meta-level" activities include gesturing and annotating, and are used to communicate information about the underlying artifacts in the conference.

GroupKit provides support for these activities using overlays. Conceptually, overlays act as transparent windows, and can be placed on top of any conference glyph (or in fact, on top of other overlays). Overlays draw on top of the conference glyph, and handle mouse events before passing them on to the underlying conference glyph. Two overlays are provided, one for graphical annotation (sketching), and one for gesturing (using multiple cursors). Other overlays could be designed, derived from the GroupKit "Overlay" class.

The SketchPad Overlay

The first overlay allows annotating using simple freehand bitmap drawing with the mouse. All annotations created are immediately transmitted to other conference users.

Though the Sketchpad can overlay any conference glyph, for the sake of example an "empty" glyph can be created.

```
class EmptyGlyph : public ConferenceGlyph {
public:
  EmptyGlyph( Style*, Conference*, int, int);
};

EmptyGlyph::EmptyGlyph(Style* style, Conference* conf,
                       int x, int y) :
    ConferenceGlyph(nil, style, conf)
{
  LayoutKit& layout = *LayoutKit::instance();
  WidgetKit& kit = *WidgetKit::instance();
  body(
    new Background(
      layout.vbox_first_aligned(
        layout.vspace(y),
        layout.hspace(x)
      ),
      kit.background()
    )
  );
}
```

A Sketchpad overlay could be placed over the EmptyGlyph as follows.

```
int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("SketchDemo",
       argc, argv);
  Conference* confer = session->conference();
  WidgetKit& kit = *WidgetKit::instance();
  LayoutKit& layout = *LayoutKit::instance();
  session->run_window(
    new ApplicationWindow(
      kit.inset_frame(
        new Sketchpad(
          new EmptyGlyph( session->style(), confer, 300, 300)
          , session->style(), confer
        )
      )
    )
  );
}
```

The Cursor Overlay

The Cursor Overlay allows communicating gestural information via multiple cursors that appear on all displays. This allows pointing to objects underneath the overlay.

Working with the Cursor Overlay is just like working with the Sketchpad overlay. The following fragment combines both the overlays (see Appendix A for a complete listing).

```
  session->run_window(
    new ApplicationWindow(
      kit.inset_frame(
        new CursorOverlay(
          new Sketchpad(
            new EmptyGlyph( session->style(), confer,300,300)
            , session->style(), confer
          )
          , session->style(), confer
        )
      )
    )
  );
```

Monitoring the User List

Monitoring

GroupKit allows objects (such as a ConferenceGlyph subclass) to be notified when users enter or leave the conference. This is handy for maintaining data structures for each conference user. This section describes how to do this, using the example of a glyph which simply monitors the users in a conference, displaying who is currently in the conference, and information about each users.

Various details, particularly to do with updating data structures will be omitted here for the sake of clarity. A complete listing of the program is available in Appendix A.

A UserMonitor class is defined, which is a subclass of both ConferenceGlyph and also ConferenceMonitor. ConferenceMonitor is a

virtual class which can be notified about users entering or leaving a conference.

```
class UserMonitor : public ConferenceMonitor,
                    public ConferenceGlyph {
public:
  UserMonitor( WidgetKit*, Conference* );
  virtual void newUser(AttributeList*);
  virtual void userLeaving(int id);
// various glyphs for the display
};
```

The constructor defines the interface, in this case a message field (giving the last person who entered or left the conference), a string browser with the list of users, and a string browser giving information (all available attributes) for the user selected in the top browser. Two tables are used, one to hold the attribute lists for each user, and one to indicate which user is on what line in the top string browser.

The last line registers the UserMonitor with the list of monitors in the Conference, so it will be informed when users enter or leave the conference.

```
UserMonitor::UserMonitor(WidgetKit* kit, Conference* conf) :
        ConferenceGlyph(nil, kit->style(), conf)
{
  LayoutKit& layout = *LayoutKit::instance();
  msg_ = new NoEditField(" ", kit );
  users_ = new StringBrowser(kit, nil,
           new ActionCallback(UserMonitor)(this,
                              &UserMonitor::selectUser));
  info_ = new StringBrowser(kit, nil, nil);

  patch_ = new Patch(
    kit->inset_frame(
      layout.margin(
        layout.vbox(
          msg_,
          new LabelledScrollList( kit, " ", users_,200,100),
          new LabelledScrollList( kit, " ", info_, 200, 100)
        )
      , 10.0)
    )
  );
  body(patch_);
  attrs_ = new AttrListTable(20);
  lines_ = new IntTable(20);
  conf->monitors()->append(this);
}
```

New Users

When new users enter the conference, the routine "newUser" in any ConferenceMonitors registered with the Conference is called. The routine is passed an attribute list for the user. The UserMonitor extracts the user number (as a key for its data structures) as well as the name, for display in the top string browser, and the top message field. The whole list is stored for use in the "selectUser" routine, described below.

```
void UserMonitor::newUser( AttributeList* al ) {
  char id[80], name[80], s[80];
  al->find_attribute( "usernum", id );
  al->find_attribute( "username", name );
  users_->Append( name );
  lines_->insert( users_->Count() - 1, atoi(id) );
  attrs_->insert( atoi(id), al );
  sprintf(s, "%s has entered.", name);
  msg_->field(s);
  patch_->redraw();
}
```

The selectUser routine is called when the user clicks on one of the conference users in the top browser. It displays the attribute list for the selected user in the lower browser.

```
void UserMonitor::selectUser() {
  AttributeList* al;
  int id;
  char s[100];
  info_->Clear();
  if( users_->selected() >= 0)
    if (lines_->find( id, users_->selected() ))
      if (attrs_->find(al, id))
        for (ListItr(AVPairList) i(*al->list()); i.more();
                        i.next())
        {
          sprintf(s, "%s: %s", i.cur()->attr, i.cur()->val);
          info_->Append(s);
        }
  patch_->redraw();
}
```

Users Leaving

As users leave the conference, the "userLeaving" routine is called, passing the id number of the user as a parameter. The UserMonitor updates the two internal tables (not shown here), deletes the user from the string browser and writes a message saying the user has left.

```
void UserMonitor::userLeaving( int id ) {
  char s[80], name[80];
  AttributeList* al;
  int line;

// update the tables and set line to be the line in the
// browser the user name was on
  users_->remove( line );

  if( attrs_->find( al, id )) {
    al->find_attribute( "username", name );
    sprintf( s, "%s has left.", name );
    msg_->field(s);
    attrs_->remove( id );
  }
  patch_->redraw();
}
```

Defining New Registration Methods

Why Define New Registration Methods?

Different groups will work in different ways. Its important that the software allow the group to work in ways that are natural for them. One of the things that varies from group to group is how users are allowed to create, enter and leave conferences. For example, some groups may have meetings where anyone is allowed to join conferences. Others may have meetings where new users can join only if they are on a specific list, or are "approved" by a user already in the conference.

GroupKit allows developers to define new registration methods using its "open protocols" feature (for more details, see the CSCW paper). This section briefly describes the process a developer would go through in order to create a new registration method.

The distribution includes three registrar clients supporting three different registration methods. The first is the "startup" client seen earlier, which provides an "open" registration policy, where any user can join any conference. The second is a "master" client (src/examples/reg-master) which is suitable for use by a facilitator in a strictly facilitated conference (controls what applications other users will see). Finally, a "slave" client (src/examples/reg-slave) provides a client for a user in a facilitated conference, where it is expected a facilitator will control the conferences the user will see. Although the last two programs are not complete examples, they provide useful information on how one could create such registration methods.

Creating a New Registration Semantics

The first step is deciding exactly what the registration method you would like should do. Using the master / slave clients mentioned earlier, we might consider the following:

Master:

- can create any conference
- will join to all conferences we create
- can delete any conference we are managing
- can join any user in the facilitated conference to a conference we are managing
- can remove any user from a conference we are managing
- can remove any user fromthe facilitated conference

Slave:

- will initially join a single facilitated conference
- on receiving a "new-user" message from the registrar specifying ourself, we will create the conference
- on receiving a "delete-user" message from the registrar specifying ourself, we will delete the conference
- on receiving a "delete-user" message from the registrar specifying ourself and where the conference specified is the original facilitated conference we will quit

Coding the Registrar Client

These semantics must then be translated into code for the registrar client.

The following code fragment might be used by the master facilitated registrar client to add a user to an existing conference (for example a brainstorming tool used in the facilitated conference).

```
Master::join_user(int confnum, int usernum) {

  char s[80];
  AttrListTable* fc_usrs;
  AttributeList* user;

// first, find the user in the overall facilitated conference
// so we can use their host and port number, etc.

  users_tbl_->find( fc_usrs, confnum_ ));
  fc_usrs->find( user, usernum );

// copy the attribute list, but change the conference number
// from that of the facilitated conference to that of the
// subconference (e.g. brainstorming tool)

  sprintf(s, "%d", confnum);
  user->attribute( "confnum", s);

// now send the new-user message to the registrar

  callJoinConference( user );
}
```

The slave registrar client would receive a "new-user" message from the registrar, and might handle it as follows:

```
void Slave::foundNewUser(AttributeList* al) {

  char conf[80], usernum[80], hostname[80], port[80],
       hostnm[80], portnm[80];
  AttrListTable* usrs;
  AttributeList* conf_al;

// extract info from attribute list and insert the new
// user into the users_tbl_ for the conference

  al->find_attribute("confnum", conf);
  al->find_attribute("usernum", usernum);
  al->find_attribute("host", hostname);
  al->find_attribute("port", port);
  if (users_tbl_->find(usrs, atoi(conf))
    usrs->insert( atoi(usernum), al );

// check if we're the user

  if ( (strcmp(hostname, GroupSession::host_name()==0) &&
       ( atoi(port) == lPort() )) {

// create the conference and join to it

    conference_tbl_->find( conf_al, atoi(conf));
    coord_->createConference( conf_al );
    coord_->setLocalInfo ( al );

// join to all the other existing users in the conference
// (code omitted here)

  }
}
```

Building an Interface

Finally, an interface for the new registrar client must be constructed. The master program provides an example of how one such interface could be constructed and linked to the registrar client.

Appendix A. Program Listings

Deck Flip Program

DeckFlip.h – Header file for Conference Glyph

```
#ifndef __deckflip_h
#define __deckflip_h

#include <gk/confglyph.h>

class DeckFlip : public ConferenceGlyph {
public:
  DeckFlip( class Style*, class Conference* );
private:
  void prev();
  void next();
  void flip(GlyphIndex cur);
  void real_flip(GlyphIndex cur);
  void remoteFlip(char *);
  class Patch* patch_;
  class Deck* deck_;
};

#endif
```

DeckFlip.c – Source file for Conference Glyph

```
declareActionCallback(DeckFlip)
implementActionCallback(DeckFlip)

declareStrActionCallback(DeckFlip)
implementStrActionCallback(DeckFlip)


#define DEMOCALLBACK  200

/*****************************************************************************
 *
 * construct the glyph, set up callbacks, etc.
 *
 *****************************************************************************/

DeckFlip::DeckFlip(Style* style, Conference* conf) :
        ConferenceGlyph(nil, style, conf)
{
  WidgetKit& kit = *WidgetKit::instance();
  const LayoutKit& layout = *LayoutKit::instance();

  deck_ = layout.deck();
  deck_->append(kit.label("Hi mom!"));
  deck_->append(kit.label("Big Bird"));
  deck_->append(kit.label("Oscar"));
  deck_->flip_to(0);
```

```
  patch_ = new Patch(deck_);
  body(
    kit.inset_frame(
      layout.margin(
        layout.vbox(
          patch_,
          layout.vglue(5.0),
          layout.hbox(
            kit.push_button("Next", new ActionCallback(DeckFlip)
                  (this, &DeckFlip::next)),
            layout.hglue(10.0),
            kit.push_button("Previous", new ActionCallback(DeckFlip)
                  (this, &DeckFlip::prev))
          ),
        ),
        10.0
      )
    )
  );
  conf->connections()->callbacks()->insert( DEMOCALLBACK,
           new StrActionCallback(DeckFlip)(this, &DeckFlip::remoteFlip));
}

/*****************************************************************************
 *
 * callbacks when user presses previous or next buttons
 *
 *****************************************************************************/

void DeckFlip::prev() {
  GlyphIndex cur = deck_->card() - 1;
  flip( cur == -1 ? deck_->count() - 1 : cur );
}

void DeckFlip::next() {
  GlyphIndex cur = deck_->card() + 1;
  flip ( cur == deck_->count() ? 0 : cur );
}

/*****************************************************************************
 *
 * call real_flip to change the glyph and also tell other users
 *
 *****************************************************************************/

void DeckFlip::flip(GlyphIndex cur) {
  char s[10];
  real_flip(cur);
  sprintf(s, "%d", cur);
  conference()->connections()->toAll( new StrMsgSender( DEMOCALLBACK, s) );
}

/*****************************************************************************
 *
 * change glyph to appropriate card
 *
 *****************************************************************************/

void DeckFlip::real_flip(GlyphIndex cur) {
  deck_->flip_to(cur);
  patch_->redraw();
}

/*****************************************************************************
```

```
 *
 * remote user changed the card
 *
 ************************************************************************/

void DeckFlip::remoteFlip(char* s) {
  int which;
  sscanf(s, "%d", &which);
  real_flip(which);
}
```

DeckEg.c – Main Program

```
#include <gk/groupsession.h>
#include <InterViews/window.h>
#include <stdio.h>
#include "deckflip.h"

int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("DeckFlip", argc, argv);
  session->run_window(
    new ApplicationWindow(
      new DeckFlip(session->style(), session->conference())
    )
  );
}
```

UserMonitor Program

UserMon.h – Header file for conference glyph

```
#ifndef __usermon_h
#define __usermon_h

#include <gk/confmonitor.h>
#include <gk/confglyph.h>

class UserMonitor : public ConferenceMonitor, public ConferenceGlyph {
public:
  UserMonitor( class WidgetKit*, class Conference* );
  virtual void newUser(class AttributeList*);
  virtual void userLeaving(int id);
protected:
  void selectUser();
  class StringBrowser* users_;
  class NoEditField* msg_;
  class Patch* patch_;
  class StringBrowser* info_;
  class AttrListTable* attrs_;
  class IntTable* lines_;
};


#endif
```

UserMon.c – Source for Conference Glyph

```
#include "usermon.h"
#include <InterViews/patch.h>
#include <gk/attrlist.h>
#include <gk/conference.h>
```

```
#include <gk-ui/strbrowser.h>
#include <gk-ui/noeditfield.h>
#include <IV-look/kit.h>
#include <InterViews/layout.h>
#include <InterViews/action.h>
#include <gk-ui/labscrlst.h>
#include <OS/table.h>
#include <stdio.h>

declareActionCallback(UserMonitor)
implementActionCallback(UserMonitor)

declareTable(IntTable, int, int)
implementTable(IntTable, int, int)

/*****************************************************************************
 *
 * Create a user monitor, consisting of a list of users, and underneath
 * that a list holding the attributes for the selected user, as well as
 * a static text field holding messages
 *
 *****************************************************************************/

UserMonitor::UserMonitor(WidgetKit* kit, Conference* conf) :
          ConferenceGlyph(nil, kit->style(), conf)
{
  users_ = new StringBrowser(kit, nil, new ActionCallback(UserMonitor)
                             (this, &UserMonitor::selectUser));
  info_ = new StringBrowser(kit, nil, nil);
  msg_ = new NoEditField(" ", kit );
  LayoutKit& layout = *LayoutKit::instance();
  patch_ = new Patch(
    kit->inset_frame(
      layout.margin(
        layout.vbox(
          msg_,
          new LabelledScrollList( kit, " ", users_, 200, 100),
          new LabelledScrollList( kit, " ", info_, 200, 100)
        )
      , 10.0)
    )
  );
  body(patch_);
  attrs_ = new AttrListTable(20);
  lines_ = new IntTable(20);
  conf->monitors()->append(this);
}


/*****************************************************************************
 *
 * A user has been selected - display their attributes in the lower browser
 *
 *****************************************************************************/

void UserMonitor::selectUser() {
  AttributeList* al;
  int id;
  char s[100];
  info_->Clear();
  if( users_->selected() >= 0)
    if (lines_->find( id, users_->selected() ))
      if (attrs_->find(al, id))
        for (ListItr(AVPairList) i(*al->list()); i.more(); i.next()) {
```

```
            sprintf(s, "%s: %s", i.cur()->attr, i.cur()->val);
            info_->Append(s);
          }
  patch_->redraw();
}


/*****************************************************************************
 *
 * a new users has joined, stick their name in the users browser, and
 * place a message in the static text field
 *
 *****************************************************************************/

void UserMonitor::newUser( AttributeList* al ) {
  char id[80], name[80], s[80];
  al->find_attribute( "usernum", id );
  al->find_attribute( "username", name );
  users_->Append( name );
  lines_->insert( users_->Count() - 1, atoi(id) );
  attrs_->insert( atoi(id), al );
  sprintf(s, "%s has entered.", name);
  msg_->field(s);
  patch_->redraw();
}


/*****************************************************************************
 *
 * a user is leaving, remove them from the users browser and place a message
 * in the static text field.  have to muck with the table so that every
 * user who was below the removed one gets moved up a line
 *
 *****************************************************************************/

void UserMonitor::userLeaving( int id ) {
  char s[80], name[80];
  AttributeList* al;
  int line;
  IntTable* tbl;

  for (TableIterator(IntTable) j(*lines_); j.more(); j.next())
    if (j.cur_value() == id)
      line = j.cur_key();
  fprintf(stderr, "line is %d\n", line);
  tbl = new IntTable(20);
  for (TableIterator(IntTable) i(*lines_); i.more(); i.next())
    if (i.cur_key() > line)
      tbl->insert( i.cur_key()-1, i.cur_value());
    else if (i.cur_key() < line)
      tbl->insert( i.cur_key(), i.cur_value());
  delete lines_;
  lines_ = tbl;
  users_->remove( line );

  if( attrs_->find( al, id )) {
    al->find_attribute( "username", name );
    sprintf( s, "%s has left.", name );
    msg_->field(s);
    attrs_->remove( id );
  }
  patch_->redraw();
}
```

Mon.c – main program

```
#include <gk/groupsession.h>
#include <InterViews/window.h>
#include <stdio.h>
#include "usermon.h"
#include <IV-look/kit.h>

int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("UserMonitor", argc, argv);
  session->run_window(
    new ApplicationWindow(
      new UserMonitor(WidgetKit::instance(), session->conference())
    )
  );
}
```

GroupSketch Program

EmptyGlyph.h – Header file for Conference Glyph

```
#ifndef __emptyglyph_h
#define __emptyglyph_h

#include <gk/confglyph.h>

class EmptyGlyph : public ConferenceGlyph {
public:
  EmptyGlyph( class Style*, class Conference* , int, int);
};

#endif
```

EmptyGlyph.c – Source file for Conference Glyph

```
#include "emptyglyph.h"
#include <InterViews/background.h>
#include <IV-look/kit.h>
#include <InterViews/layout.h>
#include <InterViews/bitmap.h>
#include <InterViews/stencil.h>
#include <InterViews/color.h>


EmptyGlyph::EmptyGlyph(Style* style, Conference* conf, int x, int y) :
        ConferenceGlyph(nil, style, conf)
{
  Bitmap* bit = new Bitmap( (const void *)nil, x, y);
  for (int i=0; i < x; i++)
    for (int j=0; j < y; j++)
      bit->poke(0,x,y);
  body( new Background( new Stencil( bit, new Color(1,1,1)),
                        WidgetKit::instance()->background()));
}
```

GroupSketch.c – main program

```
#include <gk/groupsession.h>
#include <InterViews/window.h>
#include <stdio.h>
```

```
#include <gk/cursor.h>
#include <gk/sketchpad.h>
#include "emptyglyph.h"
#include <IV-look/kit.h>
#include <InterViews/layout.h>

int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("CursorDemo", argc, argv);
  Conference* confer = session->conference();
  session->run_window(
    new ApplicationWindow(
       WidgetKit::instance()->inset_frame( LayoutKit::instance()->margin(
              new CursorOverlay(
                new Sketchpad(
                   new EmptyGlyph(session->style(), confer, 300,300)
                   ,session->style(), confer
                )
                ,session->style(), confer
              )
        , 30.0))
    )
  );
}
```

INTRODUCTION

GroupKit overview

GroupKit is a toolkit for developing real-time groupware applications. It consists of a set of C++ classes and applications, providing a number of common groupware components. GroupKit programs run on Unix machines, under X Windows. GroupKit is based on the InterViews user interface toolkit, developed at Stanford University.

About this Reference Manual

This manual provides details of the C++ classes which make up GroupKit. This serves as a detailed reference to using the toolkit.

Previous Knowledge

This reference manual assumes a familiarity with C++ and InterViews, as well as a basic understanding of how to build GroupKit applications.

Remainder of this Document

The reference manual separates the GroupKit classes into a number of categories:

- Communications Support
- Callbacks
- Graphics and Overlays
- Conference Support
- Widgets
- Registration

Extending the Toolkit

The structure of the classes is designed to allow for extending by the developer (i.e. most implementation-specific instance variables for classes are declared "protected" not "private"). If a component does not provide the exact functionality required for an application, it may be straightforward to add that functionality by subclassing.

For Further Information

The "README" file in the GroupKit distribution provides details for installing GroupKit on the development system.

The "GroupKit Tutorial" provides a basic understanding of how to build applications with GroupKit.

The paper "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications" (Proc. CSCW '92, Toronto, Ontario) describes the motivation for GroupKit's design as well as its implementation.

The "InterViews Reference Manual" (from the InterViews distribution, available from Stanford University) provides details of the InterViews class library.

The document "A Not-Entirely Gentle Introduction to InterViews" (included in this distribution) provides a brief introduction to InterViews.

COMMUNICATIONS SUPPORT

Communications in GroupKit relies on the InterViews Dispatch library. Dispatch provides a simple front end to standard Unix sockets. The communications support added by GroupKit is provided by the following classes:

WRITER allows sending messages over a socket

CALLBACKRPCREADER receives messages from a socket, and calls a user-specified routine (the message-target) based upon a message-type

MSGSENDER provides a generic facility for sending messages over a number of different WRITERS

STRMSGSENDER provides a facility for sending a message made up of a message-type and an option-string over a number of different WRITERS

CONNECTION manages a WRITER and CALLBACKRPCREADER to form a bi-directional socket connection

CONNECTIONLIST manages a list of CONNECTIONS

INFOCONNECTION is a CONNECTION which can receive information about the remote socket

INFOCONNECTIONLIST manages a list of INFOCONNECTIONS

Key concepts:

Message-type is an integer specifying what message is being sent. The message-type is used by the receiver (a CALLBACKRPCREADER) to determine what routine should be called.

Option-string is a string providing any options or parameters required by the message. This will vary according to the message-type, and is interpreted by the message-target.

Message-target is the user-specified routine (typically a method of an object) which is called by the CALLBACKRPCREADER when a message with a given message-type is received.

CLASS: WRITER
CATEGORY: COMMUNICATIONS
SUPERCLASS: RPCWRITER*

A WRITER allows sending messages across a socket connection.

Synopsis:

```
#include <gk/writer.h>
```

Writer(const char* host, int port);

Create a WRITER which will initiate a socket connection to the given machine at the given port number.

Writer(int fd);
Writer(rpcstream* server);

Create a WRITER which accepts a socket connection initiated by a remote host.

void sendMsg(int message-type, char* option-string);

Send a message across the socket connection. Predefined message types can be found in `"groupkit.h"`.

CLASS: CALLBACKRPCREADER
CATEGORY: COMMUNICATIONS
SUPERCLASS: RPCREADER*

A CALLBACKRPCREADER receives socket messages and calls a user-specified routine.

Synopsis:

```
#include <gk/reader.h>
```

CallbackRpcReader(rpcstream* server, ReaderCallbackTable* callbacks = nil, int id = -1, RpcAction* close = nil);
CallbackRpcReader(int fd, ReaderCallbackTable* callbacks = nil, int id = -1, RpcAction* close = nil);

Create a CALLBACKRPCREADER based upon either an existing file descriptor or RPCSTREAM. Typically the RPCSTREAM is obtained from WRITER::SERVER(). A previously created table of callbacks for different message types may be specified (to permit sharing), or by default a new table will be created. An id number may optionally be specified, which is available for client use. Finally, a callback can be specified which will be activated if the socket connection is closed down.

void registerCallback(StrAction* callback, int message-type);

Add a callback to the callback table. This callback will be activated when a message with the indicated message-type is received over the socket.

int id();

Return the id number which was initialized in the constructor.

void closeCallback(RpcAction* close);

Provides an alternate way to set the callback which is activated if the socket connection is closed down.

---

* A "*" after a superclass indicates that the superclass is part of the InterViews library.

CLASS: MSGSENDER
CATEGORY: COMMUNICATIONS
SUPERCLASS: <NONE>

A MSGSENDER allows the same message to be sent over multiple sockets.

Synopsis:

```
#include <gk/msgsender.h>
```

void sendOn(Writer* writer) = 0;

A virtual method which is called to actually send the message. Subclasses must define this.

CLASS: STRMSGSENDER
CATEGORY: COMMUNICATIONS
SUPERCLASS: MSGSENDER

A STRMSGSENDER allows the same message to be sent over multiple sockets.

Synopsis:

```
#include <gk/msgsender.h>
```

StrMsgSender(int message-type, char* option-string);

Create a STRMSGSENDER, which can send the message specified by the message-type and option-string. Predefined message-types can be found in `"groupkit.h"`.

void sendOn(Writer* writer);

This method will send the message over the socket connection managed by the WRITER.

Usage example:

```
char s[10];
sprintf(s, "%d", current_value);
conference()->connections()->toAll( new StrMsgSender( SETCURVALUE, s) );
```

CLASS: CONNECTION
CATEGORY: COMMUNICATIONS
SUPERCLASS: <NONE>

A CONNECTION combines a WRITER and a CALLBACKRPCREADER to make a bi-directional socket connection.

Synopsis:

```
#include <gk/connection.h>
```

Connection(const char* host, int port, int id = -1, ReaderCallbackTable* callbacks = nil);
Connection(int fd, int id = -1, ReaderCallbackTable* callbacks = nil);
Connection(rpcstream* server, int id = -1, ReaderCallbackTable* callbacks = nil);

Create a CONNECTION, either connecting to a specified host and port, or accepting a connection from a file descriptor or RPCSTREAM. An id number may optionally be specified, which is available for client use. A previously created table of callbacks for different message types may be specified (to permit sharing), or by default a new table will be created.

int id_;

An id number which may be used by clients. In GroupKit this is normally used to specify the id number of the remote user of the socket.

CallbackRpcReader* reader();
Writer* writer();
ReaderCallbackTable* callbacks();

Methods for accessing the CALLBACKRPCREADER, the WRITER (e.g. for sending messages on the connection), and the READERCALLBACKTABLE used by the connection. The definition of READERCALLBACKTABLE can be found in `"reader.h"`.

CLASS: CONNECTIONLIST
CATEGORY: COMMUNICATIONS
SUPERCLASS: LIST*

A CONNECTIONLIST manages a list of CONNECTIONs with a shared READERCALLBACKTABLE.

Synopsis:

```
#include <gk/connection.h>
```

ConnectionList();

Create a new CONNECTIONLIST.

Connection* add(char* host, int port, int id = -1);
Connection* add(int fd, int id = -1);
Connection* add(rpcstream* server, int id = -1);

Create and add a new CONNECTION to the list.

long find(int id);

Search the list for a CONNECTION having the given value in its ID_ field, and return its index in the list. Return -1 if there is no such CONNECTION is found.

void sendTo(int id, MsgSender* messenger);
void toAll(MsgSender* messenger);

Send a message, either on the socket connection with the given value in its ID_ field, or on every socket connection in the list.

ConnList* list();

Return the underlying list, created using the InterViews LIST class, permitting operations such as LIST::INSERT, LIST::APPEND, etc.

ReaderCallbackTable* callbacks();

Return the table of callbacks used by the CONNECTIONLIST. The definition of READERCALLBACKTABLE can be found in `"reader.h"`.

CLASS: INFOCONNECTION
CATEGORY: COMMUNICATIONS
SUPERCLASS: CONNECTION

An INFOCONNECTION is a CONNECTION which accepts a special message containing information about the remote socket CONNECTION.

Synopsis:

```
#include <gk/infoconn.h>
```

InfoConnection(const char* host, int port, int id = -1, ReaderCallbackTable* callbacks = nil, int info-message-type, ConnAction* info-action);
InfoConnection(int fd, int id = -1, ReaderCallbackTable* callbacks = nil, int info-message-type, ConnAction* info-action);
InfoConnection(rpcstream* server, int id = -1, ReaderCallbackTable* callbacks = nil, int info-message-type, ConnAction* info-action);

Create an INFOCONNECTION, either connecting to a specified host and port, or accepting a connection from a file descriptor or rpcstream. An id number may optionally be specified, which is available for client use. A previously created table of callbacks for different message types may be specified (to permit sharing), or by default a new table will be created. The message-type of an initialization message should be specified, along with a callback to be activated when the message-type is received over the socket.

Notes:

INFOCONNECTIONS are used extensively in GroupKit, because CONNECTIONS are typically accessed by their id number (specifying the id of a remote user) in a CONNECTIONLIST. Those id numbers are not initialized when the socket is first created, so therefore they could not be easily accessed. By passing information in these initialization messages, a callback routine can interpret that information and set appropriate fields in the CONNECTION object (which is passed as a parameter to the callback routine).

CLASS: INFOCONNECTIONLIST
CATEGORY: COMMUNICATIONS
SUPERCLASS: CONNECTIONLIST

An INFOCONNECTIONLIST manages a list of INFOCONNECTIONs.

Synopsis:

```
#include <gk/infoconn.h>
```

InfoConnectionList(int info-message-type, class ConnAction* info-action);

Create a new INFOCONNECTIONLIST.

CALLBACKS

Callbacks are used in GroupKit in the same manner as in standard InterViews programming. They allow a user-defined method of a particular object to be notified as the result of a particular action. Three types of callbacks specific to GroupKit are defined:

STRACTION specifices a callback routine taking a string as a parameter

CONNACTION specifies a callback routine taking a string and a CONNECTION object as parameters

RPCACTION specifies a callback routine taking a CALLBACKRPCREADER and a file descriptor as parameters

Callbacks must be defined for each class of object accepting a callback. This is basically templates, however not all compilers support templates. GroupKit follows InterViews style of defining template behavior using preprocessor macros. To use callbacks therefore requires two extra steps:

```
declare<TYPE>ActionCallback(<CLASS>)
implement<TYPE>ActionCallback(<CLASS>)
```

The first line writes out a C++ class declaration for a callback of type <TYPE> (Str, Conn, or Rpc) where the callback activates a method in class <CLASS>. The second line writes out the body of the class, the actual definition of each of the methods for the callback. Creating a <TYPE> callback to a member function <FUNC> in an object <OBJ> of type <CLASS> is done by:

```
new <TYPE>ActionCallback(<CLASS>)(<OBJ>,&<CLASS>::<FUNC>);
```

New types of callbacks (accepting different parameters) can also be created, by making minor changes to the macros which create the current types of callbacks.

CLASS: STRACTION
CATEGORY: CALLBACKS
SUPERCLASS: RESOURCE*

A STRACTION specifies a callback routine taking a string as a parameter.

Synopsis:

```
#include <gk/straction.h>
```

StrAction();

Create a new STRACTION.

void execute(char *);

Activate the callback, specifying the string to be passed to the callback routine.

Usage example:

```
class App {
          /* various routines */
          void App:doit(char* s);   /* Interpret the string for this message-type */
};

declareStrActionCallback(App)
implementStrActionCallback(App)

App* myapp = new App();
connection->callbacks()->insert( MSGTYPE, new StrActionCallback(App)(myapp,
          &App::doit));
```

Notes:

STRACTIONS are typically used in GroupKit to specify callbacks for messages arriving over socket connections. The message-type allows the CALLBACKRPCREADER to identify the callback, and the option-string is passed to the callback routine to be interpreted.

CLASS: CONNACTION
CATEGORY: CALLBACKS
SUPERCLASS: RESOURCE*

A CONNACTION specifies a callback routine taking a string and a CONNECTION as parameters.

Synopsis:

```
#include <gk/connaction.h>
```

ConnAction();

Create a new CONNACTION.

void execute(char *, Connection*);

Activate the callback, specifying the string and CONNECTION to be passed to the callback routine.

Usage example:

```
class App {
          /* various routines */
          void App:init(char* s, Connection* c);  /* initialize the Connection */
};

declareConnActionCallback(App)
implementConnActionCallback(App)

App* myapp = new App();
conns_ = new InfoConnectionList( INFOMSG, new ConnActionCallback(App)(myapp,
          &App::init));
```

Notes:

CONNACTIONS are typically used in GroupKit with INFOCONNECTION objects, to initialize a CONNECTION object when information (such as id number) is received from the other end of the CONNECTION. The callback would receive the option-string in the string parameter, and interpret it to fill in fields in the CONNECTION object passed as the other parameter.

CLASS: RPCACTION
CATEGORY: CALLBACKS
SUPERCLASS: RESOURCE*

A RPCACTION specifies a callback routine taking a CALLBACKRPCREADER and a file descriptor as parameters.

Synopsis:

```
#include <gk/rpcaction.h>
```

RpcAction();

Create a new RPCACTION.

void execute(CallbackRpcReader*, int fd);

Activate the callback, specifying the CALLBACKRPCREADER and file descriptor to be passed to the callback routine.

Usage example:

```
class App {
          /* various routines */
          void App:closing(CallbackRpcReader*, int fd);     /* Clean up socket */
};

declareRpcActionCallback(App)
implementRpcActionCallback(App)

App* myapp = new App();
conn_->reader()->closeCallback( new RpcActionCallback(App)(myapp, &App::close));
```

Notes:

RPCACTIONS are typically used in GroupKit to trap closed socket connections. A CONNECTIONLIST for example uses an RPCACTION so that when one of its sockets closes it can find it and remove it from the list.

GRAPHICS AND OVERLAYS

GroupKit relies on the InterViews glyph mechanism as a base for its user interfaces. Glyphs are lightweight objects composed using a TeX "boxes and glue" strategy. Several GroupKit glyphs are defined:

CONFERENCEGLYPH is the primary building block for GroupKit user interfaces, referencing a Conference object so that its methods can send and receive messages

OVERLAY is a base class allowing the construction of glyphs which overlay other ConferenceGlyphs

SKETCHPAD is an overlay which allows multi-user bitmap sketching or annotation on top of other glyphs

CURSOROVERLAY is an overlay which allows gesturing via multiple cursors on top of other glyphs

CLASS: CONFERENCEGLYPH
CATEGORY: GRAPHICS AND OVERLAYS
SUPERCLASS: ACTIVEHANDLER*

A CONFERENCEGLYPH is a GLYPH containing a reference to a CONFERENCE object.

Synopsis:

```
#include <gk/confglyph.h>
```

ConferenceGlyph(Glyph*, Style*, Conference*);

Create a CONFERENCEGLYPH.

Conference* conference();

Return the CONFERENCE object associated with the CONFERENCEGLYPH.

Notes:

A CONFERENCEGLYPH is the basis for building the graphical components of GroupKit programs. The GLYPH in the constructor specifies the graphics (as in a MONOGLYPH), and can also be assigned via MONOGLYPH::BODY(GLYPH*).

Usage example:

This example shows a CONFERENCEGLYPH subclass containing a single string label, passed as a parameter to the glyph. When the glyph is pressed (which is detectable because a CONFERENCEGLYPH is an ACTIVEHANDLER), labels for all CONFERENCE users are set to the string of the user who pressed the button The PATCH is used in order to force a redraw of the glyph when the string changes. The constructor uses the associated CONFERENCE for sending messages, as well as establishing callbacks for messages.

```
class ShowString : public ConferenceGlyph {
public:
    ShowString( char* str, Style*, Conference* );
    void Change( char* str );
    virtual void press();
    Patch* patch_;
};

ShowString::ShowString( char* str, Style* style, Conference* conf) :
          ConferenceGlyph(nil, style, conf)
{
  patch_ = new Patch ( WidgetKit::instance()->label( str ) );
  body( patch_ );
  conference()->connections()->callbacks()->insert( STRINGMSG,
          new StrActionCallback(ShowString)(this, &ShowString::Change));
}

void ShowString::press()
{
  conference()->connections()->toAll( new StrMsgSender( STRINGMSG, str) );
}

void ShowString::Change(char* str)
{
  patch_->body( WidgetKit::instance()->label( str ) );
  patch_->reallocate();
  patch_->redraw();
}
```

CLASS: OVERLAY
CATEGORY: GRAPHICS AND OVERLAYS
SUPERCLASS: ACTIVEHANDLER*

An OVERLAY permits one layer of graphics to be placed on top of another.

Synopsis:

```
#include <gk/overlay.h>
```

Overlay(ActiveHandler* inside, class Style* style, class Conference* conference);

Create an OVERLAY on top of the specified ACTIVEHANDLER. The default OVERLAY will essentially do nothing, so subclasses will add specific functionality, arranging the graphics using LAYOUTKIT::OVERLAY for example.

void move(const Event&e);
void press(const Event&e);
void drag(const Event&e);
void release(const Event&e);
void keystroke(const Event&e);
void double_click(const Event&e);
void enter();
void leave();

Handle events. The default behavior is to call the appropriate method of the "inside" glyph specified in the constructor. Subclasses can override, but should call the appropriate method in OVERLAY if the event should also be passed on to the inside glyph.

Conference* conference();

Return the CONFERENCE object associated with the OVERLAY.

[protected] void pick(Canvas* c, const Allocation& a, int depth, Hit& h);

The OVERLAY::PICK routine is like the normal ACTIVEHANDLER::PICK except that it will not call pick on the subglyphs (i.e. the "inside" glyph). This ensures that the OVERLAY, and not the leaf glyphs will receive the events first.

[protected] boolean event(Event& e);

The OVERLAY::EVENT routine replaces the ACTIVEHANDLER's normal event processing routines. It processes the event in a similar way to the ACTIVEHANDLER, but then passes the event to the inside glyph for processing.

Notes:

Both of the provided overlays (SKETCHPAD and CURSOR) provide some basic functionality. Developers may wish to subclass these overlays to meet their actual needs. More experience working with the standard overlays in different groupware applications may suggest ways that the standard ones can be parameterized through mechanisms such as X resources.

CLASS: SKETCHPAD
CATEGORY: GRAPHICS AND OVERLAYS
SUPERCLASS: OVERLAY

A SKETCHPAD is an OVERLAY which permits freehand sketching.

Synopsis:

```
#include <gk/sketchpad.h>
```

Sketchpad(ActiveHandler* inside, class Style* style, class Conference* conference);

Create a new SKETCHPAD overlay. A LAYOUTKIT::OVERLAY is used to overlay the inside glyph with a bitmap the same size as the inside glyph.

void press(const Event &e);
drag(const Event&e);
release(const Event &e);

These routines handle the drawing events from the local host, updating the bitmap, broadcasting changes to the other CONFERENCES (actually the other SKETCHPAD overlays) and then passing the events on to the inside glyph.

void doLine(int x0, int y0, int x1, int y1);

This routine draws the lines on the bitmap, and damages the canvas appropriately to generate a redraw.

void remoteScribble(char *s);

This callback routine is used to get the coordinates of lines initiated on other users' bitmaps.

Usage example:

```
int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("SketchDemo", argc, argv);
  Conference* confer = session->conference();
  session->run_window(
    new ApplicationWindow(
      new Sketchpad(
          new PictureConfGlyph(session->style(), confer),
          session->style(), confer
      )
    )
  );
}
```

Notes:

The following resource (shown here with default value) is currently defined:

*SketchOverlay–foreground:	black	(color of bitmap annotations)

The SKETCHPAD will eventually use the following resources to specify its behavior:

*SketchOverlay–lineWidth:	1	(width of annotations)
*SketchOverlay–button:	1	(which button for drawing)

The SKETCHPAD relies on a fixed-size bitmap, meaning that it cannot be resized.

This overlay should be extended to support at the very least freehand erasing (using another mouse button) and also a "Clear" operator to erase the entire bitmap.

CLASS: CURSOROVERLAY
CATEGORY: GRAPHICS AND OVERLAYS
SUPERCLASS: OVERLAY, CONFERENCEMONITOR

A CURSOROVERLAY permits gesturing using multiple cursors over another glyph.

Synopsis:

```
#include <gk/cursor.h>
```

CursorOverlay(ActiveHandler* inside, class Style* style, class Conference* conference);

Create a new CURSOROVERLAY. A PAGE is used to place cursor bitmaps over the inside glyph.

void move(const Event& e);
void drag(const Event& e);
void enter();
void leave();

These routines handle events from the local host, moving the local cursor, broadcasting changes to the other CURSOROVERLAYS, and then passing the events to the inside glyph.

void moveCursor(int id, int x, int y);

This routine actually moves one of the cursors (specified by id number of its associated user) to the appropriate spot on the PAGE.

void remoteMove(char *msg);

This callback routine is used to get the coordinates of one of the remote users' cursors.

void newUser(AttributeList* user_attrs);
void userLeaving(int id);

These two routines are called to indicate users entering or leaving the CONFERENCE. Cursors in the PAGE are added or removed, and internal data structures are updated appropriately.

Usage example:

```
int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("CursorDemo", argc, argv);
  Conference* confer = session->conference();
  session->run_window(
    new ApplicationWindow(
      new CursorOverlay(
          new PictureConfGlyph(session->style(), confer),
          session->style(), confer
      )
    )
  );
}
```

Notes:

The following resources (shown here with default values) are currently used:

*CursorOverlay–foreground:	black	(color of bitmap cursors)
*CursorOverlay-localBitmapCursor:	off	(display bitmap cursor for local user)
*CursorOverlay–localRealCursor:	on	(use the X cursor for local user)

The CursorOverlay will eventually use the following resources to specify its behavior:

*CursorOverlay–bitmap:	cursor.bit	(bitmap used for cursors)
*CursorOverlay–annotateWithNames:	off	(display user's name under bitmap)

CONFERENCE SUPPORT

GroupKit applications are called conferences. Facilities are provided in GroupKit to minimize the work needed to maintain conferences. The center of this is the **CONFERENCE** object. It provides facilities for:

- accepting new users into the conference
- maintaining a list of users in the conference
- sending messages to one or more users in the conference
- setting up callbacks for messages received
- notifying other objects when users enter or leave the conference

Much of the application specific functionality tends to reside in the user interface portions of the application (in CONFERENCEGLYPHS). These glyphs will use the facilities provided by the CONFERENCE object for their communications needs. If these glyphs need to know about users entering or leaving the conference, they can be subclassed from **CONFERENCEMONITOR**, and register themselves with the CONFERENCE.

The CONFERENCE objects rely on the registration components for information about users joining and leaving the conference. The **COORDINATOR** serves as an interface between the registration components and one or more CONFERENCES. It will create one or more CONFERENCES (at the request of the registration system) that all share a common registration system.

As well, GroupKit applications require a number of parameters to control how they behave (e.g. names of conferences, host and port numbers of different components). These parameters come from X11 resources and command line arguments. The **GROUPSESSION** object augments the standard InterViews SESSION to handle GroupKit specific parameters.

Finally, a number of support classes help maintain information throughout GroupKit, both for internal use and transmission. An **AVPAIR** holds a single attribute, value pair, while an **ATTRIBUTELIST** holds a list of AVPAIRS. A table of ATTRIBUTELISTS (indexed by an integer id number) can be stored in a **ATTRLISTTABLE**, while tables of these tables (useful for holding all the users in all the conferences for example) can be stored in a **USERLISTTBL**.

CLASS: CONFERENCE
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: RPCPEER*

A CONFERENCE maintains connections to other users, as well as the user's COORDINATOR, serving as the center of communications for GroupKit applications.

Synopsis:

```
#include <gk/conference.h>
```

Conference(char* name, const char* coord_host, int coord_port, int confnum);

Create a new CONFERENCE object, given the name of the conference, the host and port number of the COORDINATOR which initiated the CONFERENCE, and the id number of the CONFERENCE. The CONFERENCE initiates a CONNECTION to the COORDINATOR in order to receive registration information, such as announcements of new users. A CONFERENCE can also be created using the GROUPSESSION object.

void newUser(AttributeList* user_attrs);
void userLeaving(int id);

These routines are called whenever a new user joins the CONFERENCE, or an existing user leaves the CONFERENCE. The default behavior is to pass the notification on to a list of CONFERENCEMONITOR objects, which can take appropriate action, such as updating application data structures to reflect the new users.

boolean createReaderAndWriter(const char* host, int port);
void createReaderAndWriter(int fd);

These routines are used when initiating or accepting new socket connections (to the COORDINATOR and from other CONFERENCE objects respectively).

void info_callback(char* msg, class Connection* c);

This routine is used as the target of the INFOCONNECTION, interpreting information about the remote CONFERENCE user and storing it in the CONNECTION object.

void connectTo(char* msg);

This routine is activated as a callback from the COORDINATOR, and initiates a CONNECTION to a new CONFERENCE user.

void youAreID(char* msg);

This routine is activated as a callback from the COORDINATOR, and provides the CONFERENCE object with information such as the id number, etc. After this information is received, the NEWUSER method is called, so care should be taken so as not to "add" the local user twice.

void removeUser(char *msg);

This routine is activated as a callback from the COORDINATOR, and signifies a user is leaving. The default behavior is to call USERLEAVING.

void deleteConference(char *msg);

This routine is activated as a callback from the COORDINATOR, signifying that the CONFERENCE should be destroyed.

int localID();
InfoConnectionList* connections();
ConferenceMonitorList* monitors();

These three routines allow access to some of the CONFERENCE object's internal data structures: the id number of the CONFERENCE, the list of CONNECTIONS to other CONFERENCE objects (useful for sending messages), and the list of CONFERENCEMONITORS (which can be manipulated using methods such as LIST::APPEND()).

CLASS: CONFERENCEMONITOR
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: <NONE>

A CONFERENCEMONITOR can be notified when users enter or leave a CONFERENCE.

Synopsis:

```
#include <gk/confmonitor.h>
```

virtual void newUser(AttributeList* user_attrs) =0;
virtual void userLeaving(int id) = 0;

These methods are called when users enter or leave the CONFERENCE.

Notes:

Typically a CONFERENCEMONITOR is used as one of the base classes (along with a CONFERENCEGLYPH perhaps) for a new class that is interested in knowing when users enter or leave the CONFERENCE. To register a CONFERENCEMONITOR with a given CONFERENCE (from the CONFERENCEMONITOR constructor) the following code might be used:

```
conference->monitors()->append(this);
```

CLASS: COORDINATOR
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: RPCPEER*

A COORDINATOR acts as an intermediary between the registration system and one or more CONFERENCEs.

Synopsis:

```
#include <gk-reg/coordinator.h>
```

Coordinator(Style* style);

Create a new Coordinator.

virtual void createReaderAndWriter(int fd);

Accept a connection from one of the CONFERENCE objects we spawned. A callback is set up so the Conference can send us information to identify itself.

virtual boolean createConference(char* name, char* type, int id);

Create a new CONFERENCE, in response to a request from the registration mechanisms. The type of CONFERENCE to create is specified by the second parameter, which maps to a conference description resource. Resources are specified in the following manner:

```
*GroupKitBinDir:          <location of GroupKit binaries>
*conferenceTypes:         2
*conf1-desc:              Group Sketchpad
*conf1-prog:              gs
*conf2-desc:              Brainstorming Tool
conf2-prog:               bstorm
```

The command to start the conference is found by concatenating the `GroupKitBinDir` resource with the appropriate `conf*-prog` resource, determined by searching the `conf*-desc` resources for a match with the type parameter. Command line options are appended which specify the name and id number of the conference, as well as the local host and port number for the COORDINATOR (used so the CONFERENCE can connect to us when it is created). This command line is then used to spawn a new process to run the CONFERENCE. A sample command line might be:

```
/home/grouplab/bin/SUN4/bstorm -confname 'Ideas' -confid 23 -coordhost janu
       -coordport 1500
```

void deleteConference(int id);

Delete the specified CONFERENCE, by passing on the request to the appropriate CONFERENCE object.

void deleteUser(int conf-id, int user-id);

Delete the specfied user from the specfified CONFERENCE, by passing on the request to the appropriate CONFERENCE object.

void setLocalInfo(int conf-id, int user-id, char* userid, char* name);

Information for one of our CONFERENCES has been received, which we pass on to the CONFERENCE object itself.

void joinTo(int conf-id, int user-id, char* userid, char* name, char* host, int port);
void addrReq(char *s, class Connection*);
void addrResp(char *s);

The JOINTO routine specifies that one of our CONFERENCES should connect to the CONFERENCE run by the indicated user. However, the host and port number specify the address of the user's REGISTRARCLIENT, not the CONFERENCE itself (since the REGISTRAR only stores information on the REGISTRARCLIENT). To get around this, the JOINTO routine initiates a connection to the specified REGISTRARCLIENT to find out the CONFERENCE's

host and port number. The REGISTRARCLIENT delegates this request to its attached COORDINATOR, specifically the ADDRREQ method. The ADDRREQ method in the remote COORDINATOR looks up the host and port number of the CONFERENCE and sends it back to the local COORDINATOR. This information is received by the ADDRRESP routine, which then passes on all the required information to the CONFERENCE object to connect to the remote CONFERENCE.

RegistrarClient* rc_;

A pointer back to our REGISTRARCLIENT, mainly used to inform the REGISTRARCLIENT when one of our attached CONFERENCES dies.

Notes:

Because the CONFERENCE objects are running as separate processes which need to connect to the COORDINATOR, there is some time between when a CONFERENCE is created and the COORDINATOR can send it messages. As a result, all messages to the CONFERENCE are queued up before the CONFERENCE has hooked up, and these are sent as soon as the CONFERENCE does hook up.

CLASS: GROUPSESSION
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: SESSION*

A GROUPSESSION is a SESSION which helps manage a GroupKit application.

Synopsis:

```
#include <gk/groupsession.h>
```

GroupSession(const char* name, int& argc, charargv, const OptionDesc* = nil, const PropertyData* = nil);**

Create a new GROUPSESSION, specifying the name used for looking up resources, command line arguments, and any user-defined properties (resources) or options (mappings of command line options onto resources). The GROUPSESSION augments these with standard InterViews properties and options as well as GroupKit specific ones.

Conference* conference();

Create a new CONFERENCE object, based upon information which should be present in the command line arguments.

static const char* host_name();

Return the name (including internet domain) of the host the program is running on.

Usage example:

```
static PropertyData props[] = {
  { "*width", "750" },
  { "*height", "300" },
  { nil }
};

static OptionDesc options[] = {
  { "width=", "*width", OptionValueAfter },
  { "height=", "*height", OptionValueAfter },
  { nil }
};

int main(int argc, char** argv) {
  GroupSession* session =
              new GroupSession("BrainStorm", argc, argv, options, props);
  session->run_window(
    new ApplicationWindow(
      new BrainStormGlyph( session->style(), session->conference())));
}
```

CLASS: AVPAIR
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: <NONE>

An AVPAIR holds a single <attrivute,value> pair.

Synopsis:

```
#include <gk/attrlist.h>
```

AVPair(const char* attribute, const char* value);

Create a new AVPAIR.

char* attr;
char* val;

Attribute name and value.

CLASS: ATTRIBUTELIST
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: LIST*

An ATTRIBUTELIST holds a list of AVPAIRs.

Synopsis:

```
#include <gk/attrlist.h>
```

AttributeList();

Create a new ATTRIBUTELIST.

void attribute(const char* attr, const char* val);

Add a new attribute to the list, replacing any existing attributes with the same attribute name.

void remove_attribute(const char* attr);

Remove the indicated attribute from the list, if present.

boolean find_attr(const char* attr, char* value);

Given the attribute name, look up its value and store it in the second parameter. Return true if the attribute was found.

void write(char* output);

Write out a representation of the ATTRIBUTELIST, suitable for transmission, in the output string.

static AttributeList* read(const char* input);

Take an input string, presumably generated by WRITE, and rebuild the original ATTRIBUTELIST from it.

AVPairList* list();

Return the InterViews LIST used to hold the AVPAIRS.

Usage example:

```
main() {
 AttributeList* a1 = new AttributeList();
 char p[80], q[80], r[80], s[1000];

 a1->attribute("id", "123");
 a1->attribute("desc", "blah:hi");

 if(a1->find_attribute("id", p)) printf("id is <%s>\n", p);
 if(a1->find_attribute("desc", p)) printf("desc is <%s>\n", p);

 a1->attribute("id", "45");
 if(a1->find_attribute("id", p)) printf("id is <%s>\n", p);

 a1->write(s);
 printf("<%s>\n", s);

 AttributeList* a2 = AttributeList::read(s);
 if(a2->find_attribute("id", p)) printf("id is <%s>\n", p);
 if(a2->find_attribute("desc", p)) printf("desc is <%s>\n", p);
}
```

Output:

```
id is <123>
desc is <blah:hi>
id is <45>
<desc=blah\:hi:id=45>
id is <45>
desc is <blah:hi>
```

CLASS: ATTRLISTTABLE
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: TABLE*

An ATTRLISTTABLE holds a table of ATTRIBUTELISTs.

Synopsis:

```
#include <gk/attrlist.h>
```

AttrListTable(int size = 30);

Create a new ATTRLISTTABLE.

void write(char* output);

Write out a representation of the ATTRLISTTABLE, suitable for transmission, in the output string. The format is identical to that used by ATTRIBUTELIST::WRITE, with ampersands placed between adjacent ATTRIBUTELISTS.

static AttrListTable* read(const char* input, const char* key_attr, int size);

Take an input string, presumably generated by WRITE, and rebuild the original ATTRLISTTABLE from it. The KEY_ATTR parameter indicates the attribute name (present in each ATTRIBUTELIST coded in the input) used as the key for the table. The value for this attribute must be an integer. The SIZE specifies the hash table size.

Notes:

An ATTRLISTTABLE is a subclass of an ATTRLSTTBL, an InterViews TABLE holding ATTRIBUTELIST pointers indexed by an integer key, allowing all the normal TABLE operations to be performed. However, due to the way that TABLES are implemented using preprocessor macros, it is necessary to declare TABLEITERATORS over ATTRLISTTABLES as TABLEITERATOR(ATTRLSTTBL) rather than TABLEITERATOR(ATTRLISTTABLE).

CLASS: USERLISTTBL
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: TABLE*

A USERLISTTBL holds a table of ATTRLISTTABLES.

Synopsis:

```
#include <gk/attrlist.h>
```

Notes:

This class is just an InterViews Table, indexed by an integer key, and holding AttrListTables. This is convenient for holding lists of users for all the conference. Each item in this table (indexed by the conference id number) holds the table of users (indexed by user id number) for each conference.

WIDGETS

A number of general purpose widgets (InterViews glyphs) which are not part of the InterViews distribution are also included. These are used in various places within GroupKit, and can be useful building blocks for interfaces. The extra components are:

STRINGBROWSER which presents a list of strings in a box

NOEDITFIELD which presents a label in a fancy box like an edit field, but which cannot be edited

LABELLEDSCROLLLIST bundles a label, a browser, and a scrollbar, which is a common combination of glyphs

SHELL which supports running a Unix shell process inside a VT100 terminal emulator

TABULAR contains rows and columns of glyphs

As well, the DIALOGMANAGER from the InterViews sample application Doc is available. The DIALOGMANAGER allows easily creating four types of common dialog boxes:

CHOOSER is a dialog box containing a file chooser, which allows the user to select a file

ASKER is a dialog box presenting a prompt and asking the user to input a string

CONFIRMER is a dialog box presenting a prompt and asking the user to press "Yes", "No" or "Cancel"

REPORTER is a dialog box giving a message to the user which must be acknowledged

CLASS: STRINGBROWSER
CATEGORY: WIDGETS
SUPERCLASS: FILEBROWSER*

A STRINGBROWSER is a glyph which presents a list of strings in a box.

Synopsis:

```
#include <gk-ui/strbrowser.h>
```

StringBrowser(WidgetKit* kit, Action* double-click, Action* single-click);

Create a new STRINGBROWSER.

void Append(const char* string);
void Append(Glyph* g);

Append a string (or arbitrary glyph) to the end of the list of items in the STRINGBROWSER.

void Remove(int index);

Remove the indicated item from the STRINGBROWSER.

int Count();

Return the number of items in the STRINGBROWSER.

void Clear();

Remove all the items in the STRINGBROWSER.

void select(GlyphIndex index);

Select the indicated item in the STRINGBROWSER. The default behavior is to execute the single-click callback. The callback routine can figure out which item was selected using BROWSER::SELECTED.

Usage example:

```
class App : public MonoGlyph {
  App( WidgetKit* );
  void select();
  void doit();
  StringBrowser* browser_;
};

declareActionCallback(App)
implementActionCallback(App)

App::App(WidgetKit* kit) : MonoGlyph(nil) {
    browser_ = new StringBrowser( kit,
                            new ActionCallback(App)(this, &App::doit),
                            new ActionCallback(App)(this, &App::select));
    body( browser_ );
}

void App::select() { printf("something selected\n");  }
void App:doit() { printf("item %d double-clicked\n", browser_->selected()); }
```

CLASS: NOEDITFIELD
CATEGORY: WIDGETS
SUPERCLASS: FIELDEDITOR*

A NOEDITFIELD is a LABEL with a fancier appearance.

Synopsis:

```
#include <gk-ui/noeditfield.h>
```

NoEditField(const char* string, WidgetKit* kit);

Create a new NOEDITFIELD using the indicated string as its contents.

Usage example:

```
NoEditField* txt = new NoEditField("some text", WidgetKit::instance());
```

CLASS: LABELLEDSCROLLLIST
CATEGORY: WIDGETS
SUPERCLASS: MONOGLYPH*

A LABELLEDSCROLLLIST combines a browser, a scrollbar and a label.

Synopsis:

```
#include <gk-ui/labscrlst.h>
```

LabelledScrollList(WidgetKit* kit, char* lbl, FileBrowser* br, int width, int height);

Create a new LABELLEDSCROLLLIST.

Usage example:

```
StringBrowser* users = new StringBrowser( kit, nil, nil);
LabelledScrollList* users-box = new LabelledScrollList(kit, "Users", users, 200, 70);
```

CLASS: SHELL
CATEGORY: WIDGETS
SUPERCLASS: INPUTHANDLER*, IOHANDLER*

A SHELL is a GLYPH which supports running a Unix shell inside a VT100 terminal.

Synopsis:

```
#include <gk-ui/shell.h>
```

Shell(WidgetKit* kit);

Create a new SHELL, returning an 80x24 glyph which is connected by a pty to a process running a normal Unix shell.

int inputReady(int fd);

Some characters have been received on the file descriptor associated with the Unix shell process. These are interpreted and placed into the SHELL glyph.

void keystroke(const Event& e);

The user has typed a character, which is sent through the pty to the Unix shell process.

Usage example:

```
session->run_window(
  new ApplicationWindow(
    new BackGround(
      new Shell( WidgetKit::instance() ),
      WidgetKit::instance()->background()
    )
  )
);
```

Notes:

The SHELL relies on two other objects. A TEXTBUF is a MONOGLYPH which maintains the 80x24 array of characters, and provides editing operations to change the array. An EMULATOR (which was adapted from the EMULATOR found in InterViews 2.5) accepts a stream of characters (i.e. from the Unix shell process) and translates that stream into the text editing operations defined by the TEXTBUF, according to the VT100 command set.

The TEXTBUF at this time is also incomplete (i.e. some of the operations have not been implemented yet). As well it needs much performance tuning. Hoewever it does support enough functionality for most Emacs functions, etc.

CLASS: TABULAR
CATEGORY: WIDGETS
SUPERCLASS: MONOGLYPH*

A TABULAR is a GLYPH holding rows and columns of other glyphs.

Synopsis:

```
#include <gk-ui/tabular.h>
```

Tabular(int x=1, int y=1);

Create a new TABULAR, specifying the intial numbers of columns and rows.

void insert_row(int y);

Insert a new row at the given position.

void insert_col(int x);

Insert a new column at the given position.

void delete_row(int y);

Delete the row at the given position.

void delete_col(int x);

Delete the column at the given position.

void replace(int x, int y, Glyph* g);

Replace the glyph at the indicated column and row with a new glyph.

Glyph* item(int x, int y);

Return the glyph at the indicated column and row.

int rows();
int cols();

Return the number of rows or columns in the table.

CLASS: DIALOGMANAGER
CATEGORY: WIDGETS
SUPERCLASS: <NONE>

A DIALOGMANAGER allows instantiating common dialog boxes

Synopsis:

```
#include <gk-ui/DialogMgr.h>
```

DialogManager();

Create a new DIALOGMANAGER.

const char* choose(Window* win, const char* prompt, const char* filt);

Pop up a dialog box over the indicated window, displaying the prompt as well as a file chooser. The filter specifies a set of files which the file chooser should display (e.g. "*.c"). Return the name of the file selected by the user, or nil if the user hit "Cancel".

const char* ask(Window* win, const char* prompt, const char* init);

Pop up a dialog box over the indicated window, displaying a prompt and asking the user to input a string (which starts out as the value of the init parameter). Return the input string or nil if the user hit "Cancel".

int confirm(Window* win, const char* prompt);

Pop up a dialog box over the indicated window, displaying a prompt and asking the user to press "Yes", "No" or "Cancel". Return 1, 2, or 3 respectively.

void report(Window* win, const char* prompt);

Pop up a dialog box over the indicated window, displaying the prompt and asking the user to press "OK".

Usage example:

```
dlgmgr = new DialogManager();

char* s;

if ((s = (char*)dlgmgr->choose(win, "Pick a file", "*.c") != nil) {
  /* open up s and process */
}

if ((s = (char*)dlgmgr->ask(win, "Name", "Mark Roseman") != nil) {
  /* do something with s */
}

if ( dlgmgr->confirm(win, "Delete all files?") == 1) {
  /* delete the files */
}

dlgmgr->report( win, "Operation completed.");
```

Notes:

This class is copied from the InterViews sample application "Doc".

REGISTRATION

Facilities are provided in GroupKit to allow users to create, join, leave and destroy conferences. This registration system is decoupled from the conference applications using it, allowing the same registration system to be used for all conference applications.

The registration system is inherently policy free. A central **REGISTRAR** maintains a global list of all conferences and their users. The REGISTRAR accepts and obeys commands to modify these lists without regard to where the commands come from (e.g. it is perfectly acceptable for anyone to delete any user from a conference).

However, users interact with the REGISTRAR through a **REGISTRARCLIENT** (one per user, distributed across the network) which besides providing a user interface to the REGISTRAR, must also provide the semantic interpretation of any changes to the REGISTRAR's status. That is, every REGISTRARCLIENT is responsible for implementing a registration policy, as well as a user interface. The protocol by which the REGISTRAR and REGISTRARCLIENTS communicate provides the means for expressing that policy.

One sample client is currently provided, an **OPENREGCLIENT** which implements an open registration policy. Under an open registration policy, any user can create a conference, and any user can join an existing conference. Users may leave a conference they are part of, but not delete anyone else. Conferences are deleted only when the last user of the conference leaves. The OPENREGCLIENT uses a **REGCLIENTDISPLAY** to provide its user interface.

The registration system interacts with the conference applications through the COORDINATOR (described in the Conference Support section) which spawns CONFERENCES and forwards relevant registration information to them.

The registration system relies heavily on ATTRIBUTELISTS (described in the Conference Support section). The REGISTRAR accepts an ATTRIBUTELIST from its clients to describe both conferences and users. The REGISTRAR adds to these descriptions attributes for conference id number ("confnum") and user id number ("usernum"). Additionally, the COORDINATOR requires attributes to be defined for the name of the conference ("name"), the type of the conference ("type", used to look up the conference application in the X resources file), and the Internet host name ("host") and port number ("port") of various sockets. The OPENREGCLIENT also maintains information about the userid ("userid") and the full user name ("username").

CLASS: REGISTRAR
CATEGORY: REGISTRATION
SUPERCLASS: RPCPEER*

A REGISTRAR maintains global registration information for all active conferences.

Synopsis:

This class is used by the "registar" program found in src/examples/registrar.

Registrar(int port);

Create a new REGISTRAR, listening for connections from REGISTRARCLIENTS on the indicated port.

[protected] void createReaderAndWriter(int fd);

Accept a new CONNECTION from a REGISTRARCLIENT.

[protected] void new_conference(char *msg);

Add a conference (specified by an attribute list) to the internal list. Generate a unique id number for the conference.

[protected] void delete_conference(char *msg);

Delete a conference (specified by id number) from the internal list.

[protected] void disp_conference(char *msg);

Broadcast a list of current conferences to all attached REGISTRARCLIENTS.

[protected] void add_user(char *msg);

Add a user to a conference (based upon an attribute list). Generate a unique id number for the user.

[protected] void delete_user(char *msg);

Delete a user from a conference (specifying conference id number and user id number).

[protected] void display_users(char *msg);

Broadcast a list of users for a conference (specified by conference id number) to all attached REGISTRARCLIENTS.

[protected] AttrListTable* conference_tbl_;
[protected] UserListTbl* users_tbl_;

Maintain a list of all conferences and their users.

[protected] ConnectionList* connlist_;

Maintain a list of all REGISTRARCLIENTS connected to us.

Notes:

The port number the REGISTRAR should be run on is normally obtained by the "RegistrarPort" resource. The "RegistrarHost" resource specifies the machine the REGISTRAR is normally run on.

CLASS: REGISTRARCLIENT
CATEGORY: REGISTRATION
SUPERCLASS: RPCPEER*

A REGISTRARCLIENT provides an interface to the central REGISTRAR as well as defining and implementing a policy governing the semantics of different registration operations.

Synopsis:

```
#include <gk-reg/regclient.h>
```

RegistrarClient(const char* host, int port, Coordinator* coord);

Create a new REGISTRARCLIENT. Connect up to the central REGISTRAR on the given host and port. Store the COORDINATOR for later use in dealing with CONFERENCES we create.

void callJoinConference(int conf_id);
void callJoinConference(AttributeList*);

Tell the REGISTRAR the local user wishes to join the indicated conference. This routine figures out the user's name, host, etc. Alternately, let the caller provide all the host, port information.

void callLeaveConference(int conf_id, int user_id);

Tell the REGISTRAR that the user with the specified id number wants to leave the indicated conference.

void callNewConference(AttributeList*);

Tell the REGISTRAR to create a new conference, given a list of attributes (e.g. "name" and "type").

void callDeleteConference(int conf_id);

Tell the REGISTRAR to delete the indicated conference.

void PollConferences();

Tell the REGISTRAR to broadcast a list of all conferences out to the REGISTRARCLIENTS.

void PollUsers(int conf_id);

Tell the REGISTRAR to broadcast a list of users of the specified conference out to the REGISTRARCLIENTS.

void userLeft(int conf_id, int user_id) = 0;

Our COORDINATOR has told us that the indicated user has exited the indicated conference. Subclasses should handle, informing the REGISTRAR as necessary.

[protected] void foundNewConference(AttributeList*) = 0;

A new conference has been found. This routine is called not only when other users create conferences, but also when the local user creates a conference. Subclasses should handle.

[protected] void foundDeletedConference(int conf_id) = 0;

A deleted conference has been found. Subclasses should handle.

[protected] void foundNewUser(AttributeList*) = 0;

A new user for a conference has been found. This may also be the local user joining a conference. Subclasses should handle.

[protected] void foundDeletedUser(int conf_id, int user_id) = 0;

A user has been deleted from a conference. This may also include the local user. Subclasses should handle.

[protected] void UpdateConferenceList(char *info);

This routine accepts a list of current conferences from the REGISTRAR. The list is parsed, updating internal data structures (the CONFERENCELIST). This routine calls FOUNDNEWCONFERENCE and FOUNDDELETEDCONFERENCE when changes in the list are found.

[protected] void UpdateUserList(char *info);

This routine accepts a list of users of one of the conferences from the REGISTRAR. The list is parsed, updating internal data structures (one of the user lists in the CONFERENCELIST). This routine calls FOUNDNEWUSER and FOUNDDELETEDUSER when changes in the list are found.

[protected] boolean createReaderAndWriter(const char* host, int port);

This routine is used to connect up to the central REGISTRAR.

[protected] void createReaderAndWriter(int fd);

This routine is called when remote COORDINATOR objects connect to us, in search of CONFERENCE objects we've created. This is done because the central REGISTRAR maintains the host and port numbers of each user's REGISTRARCLIENT, not the CONFERENCE objects themselves. Therefore to connect to a remote CONFERENCE, a COORDINATOR first asks the REGISTRARCLIENT what the CONFERENCE object's host and port number is. This routine delegates this request to the local COORDINATOR, which maintains host and port numbers for the CONFERENCES it created.

[protected] CallbackRpcReader* reader_;
[protected] Writer* writer_;

These maintain the connection to the central Registrar.

[protected] AttrListTable* conference_tbl_;
[protected] UserListTbl* users_tbl_;

This list mirrors the tables stored in the central REGISTRAR of active conferences and their users.

[protected] Coordinator* coord_;

This points to our COORDINATOR, which actually maintains connections to and information about CONFERENCE objects that were created using this REGISTRARCLIENT.

CLASS: OPENREGCLIENT
CATEGORY: REGISTRATION
SUPERCLASS: REGISTRARCLIENT

An OPENREGCLIENT is a REGISTRARCLIENT which uses an "open" registration policy.

Synopsis:

This class is part of the "startup" program in src/examples/reg-open. Intended as an example.

OpenRegClient(const char* host, int port, Coordinator* coord);

Create a new OPENREGCLIENT.

void addDisplay(RegClientDisplay* display);

Associate a display with the REGISTRARCLIENT, which provides a user interface for working with the OPENREGCLIENT.

void userLeft(int conf_id, int user_id);

The COORDINATOR has informed us that a user has left a conference. Tell the REGISTRAR to delete the user from the conference, and also delete the conference itself if the user was the last one in it.

void foundNewConference(AttributeList*);

A new conference has been found. Update our display. If we've been waiting for that conference to be created (because we initiated its creation), tell our COORDINATOR to create it and also tell the REGISTRAR we want to join it.

void foundDeletedConference(int conf_id);

A conference has been deleted. Update the display. Tell our COORDINATOR to delete the CONFERENCE (if present).

void foundNewUser(AttributeList*);

A new user for a conference has been found. If we've been waiting to join this CONFERENCE, we should have already created the CONFERENCE object, so pass the necessary information to it via our COORDINATOR.

void foundDeletedUser(int conf_id, int user_id);

A user has left a conference. Delete the user, update the display, and then tell the COORDINATOR to delete the user from the CONFERENCE (if present).

void requestNewConference(char* name, char* type);

The user (via the display) has asked to create a new conference. Pass this on to the REGISTRAR via CALLNEWCONFERENCE.

void requestJoinConference(int conf_id);

The user (via the display) has asked to join a conference. Under this scheme we assume anyone can join any conference, so create the CONFERENCE object (via the COORDINATOR) and also forward the request to the REGISTRAR. The new CONFERENCE will also initiate connections to all existing users of the conference.

void requestLeaveConference(int conf_id);

The user (via the display) has requested to leave a CONFERENCE. Pass this request on to the REGISTRAR.

void requestViewConference(int conf_id);

The user (via the display) has asked to view the users for the indicated conference. Update the display as appropriate.

CLASS: REGCLIENTDISPLAY
CATEGORY: REGISTRATION
SUPERCLASS: MONOGLYPH*

A REGCLIENTDISPLAY provides a user interface to an OPENREGCLIENT.

Synopsis:

This class is part of the "startup" program in src/examples/reg-open.

RegClientDisplay(OpenRegClient* rc, WidgetKit* kit);

Create a new REGCLIENTDISPLAY.

void updateConference();
void updateUsers(int conf_id);

The list of conferences or users maintained by the REGISTRARCLIENT may have changed. Update our lists as appropriate.

Window* win_;

The window we're running in, needed for posting dialog boxes.

[protected] Glyph* makeGlyph();

Create the user interface for the REGCLIENTDISPLAY. This consists of: two string browsers (actually LABELLEDSCROLLLISTS) holding the list of conferences and the list of users in the selected conference; a "New" button for creating a new conference; and a "Join" button for joining an existing conference (selected in the conferences browser). The "New" button brings up a dialog box prompting the user for the name of the conference as well as its type (looked up via the resources mechanism used by the COORDINATOR for creating conferences).