SimulationGUIDebug

Brian Jacobs, Kip Nicol, Logan Rockmore

January 2009
SimPy Version: 2.0

1 SimulationGUIDebug Overview

The SimulationGUIDebug SimPy package is a tool that lets users debug SimPy programs visually. The package can be used as either a standalone debugger, or as a supplement to existing debuggers such as Python’s internal debugger, PDB. The package contains a number of user APIs that can be used to create windows for different objects within SimPy programs. The package contains a few specialized windows: EventListWindow, ProcessWindows, and ResourceWindows each which contain important default information. User defined hooks which return a str type, can be used to print out data in these windows.

2 System Requirements

SimPy 2.0

3 User Instructions

3.1 Setup

3.1.1 Registering Windows

In order for SimulationGUIDebug to create windows for your SimPy objects you must first register them. SimulationGUIDebug gives you the option to register any object or SimPy.Process subclasses. SimulationGUIDebug will create specialized windows for both the SimPy.Process and SimPy.Resource instances when they are passed to register(). If a SimPy.Process subclass is passed to register() then SimulationGUIDebug will create ProcessWindows for each instance of the class automatically once the instance is activated. The register() function also lets the user pass in an optional hook function (which returns a string) to print user defined text on the window. The register() function also lets the user pass in an optional name parameter to specify the title of the window.

register

- Call: register(obj[,hook,name])

- Parameters: obj: any object or SimPy.Process subclass class. hook: a function that returns a string. name a string to be used as the window title.
• **Return Value:** None

### 3.1.2 Specifying A Window Title

It is recommended that you give names to your Resources and Processes. SimulationGUIDebug uses the name stored in the `name` variable inside the Resource and Process classes to create the titles for the windows. The default names for a Resource is `a_resource` and the name for a Process is `a_process`. To name a process, type:

```python
Process.__init__(self, name="CarArrivals")
```

where CarArrivals is the name of the Process. The same can be done with a Resource.

### 3.1.3 Setting Run Mode

SimulationGUIDebugger runs in two modes. The first mode uses SimulationGUIDebugger’s own user prompt and steps through your simulation using SimulationGUIDebugger’s own method. The drawback to this is that you can’t run another debugger in parallel. If you wish to use another debugger, you need to set the run mode to **NO_STEP**. To do this, enter the following line of code to your simulation:

```python
SimulationGUIDebug.setRunMode(NO_STEP)
```

This will only create the windows when you run the simulation and updates them after every event. This allows you to use another debugger and still use SimulationGUIDebugger’s windows.

**setRunMode**

- **Call:** `setRunMode(runMode)`
- **Parameters:** `runMode`: SimulationGUIDebug.STEP, SimulationGUIDebug.NO_STEP
- **Return Value:** None

### 3.1.4 initialize() and simulate()

The methods of using `initialize()` and `simulate()` have not changed from their use in SimPy.

### 3.2 Execution/Operation

To start debugging, simply run your simulation as you normally would. SimulationGUIDebugger will take over from here and start by displaying the following menu:

```
[c] Continue simulation
[s] Step to next event
[b #] Add new breakpoint
[q] Quit debugger
```
Typing c will continue simulation will run your simulation until the next breakpoint or if there are no more breakpoints, it will finish your simulation. S will run the simulation until the next event and stop; updating all windows in the process. Adding a breakpoint will add a breakpoint at a given simulated time instead of line number like most debuggers. Once a breakpoint is reached, the debugger will stop the simulation at the next event after the breakpoint. Quitting the debugger will end the simulation and close all related windows.

4 Understanding The GUI

4.1 EventWindow Class

The EventWindow Class is used to create objects that control event windows. The window contains a table with the current event list data, the current time now.

The window’s table has three columns: Time and Process. The Time column contains the time at which the corresponding event is scheduled to arrive. The Process column contains the name of the process that is scheduled at its corresponding time. The Next Line column contains the line number for the next line (in code) that the corresponding process will execute.

If the scheduled event time for the top process is equal to the current time the symbol >> will be placed to the left of that process indicating that this process is currently running.

The window’s top status bar contains the current simulation time and is read: Current Time: <time>.

![Sample Event List Window](image)

Figure 1: Sample Event List Window

4.2 GenericWindow Class

The GenericWindow Class is used to create objects that control generic windows. The windows print a user defined hook method. It is subclassed by both the ProcessWindow, and the ResourceWindow.

4.3 ProcessWindow

The ProcessWindow Class is used to create objects that control process windows. The process window prints out the current status of the process: Active or Passive, the next scheduled event simulation time, interrupted status, whether the process is currently running, and a user defined hook.
4.4 ResourceWindow Class

The ResourceWindow Class is used to create objects that control resource windows. Along with the user defined hook, the resource window contains two tables:

- **ActiveQ Table**: Lists the names of all processes that are currently actively using this resource.
- **WaitQ Table**: Lists the names of all processes that are currently queuing for this resource.

5 Example

The following code, **Example.py** gives an example as to how to use SimulationGUIDebug. Important lines are explained in more detail below.

```python
# Example.py - Example for SimulationGUIDebug
from SimulationGUIDebug import *  # import SimulationGUIDebug

class SuperBeing(Process):
    def __init__(self, earth):
        Process.__init__(self)
        self.earth = earth

    def Create(self):
        while True:
```

Figure 4: Sample Resource Window

```python
yield hold, self, 1.20
man = Man(self.earth)
activate(man, man.Walk())
register(man, man.Status)  # register the man instance with hook

# Man waits for earth resource, then becomes, baby, adult, and leaves earth
class Man(Process):
    ID = 0
    def __init__(self, earth):
        Process.__init__(self, name="Man%d"%Man.ID)  # set name to ensure window title is set
        self.earth = earth
        self.status = "in heaven"
        Man.ID += 1

def Walk(self):
    self.status = "waiting for earth"
    yield request, self, self.earth
    self.status = "baby"
    yield hold, self, 1
    self.status = "adult"
    yield hold, self, 2
    self.status = "good bye earth"
    yield release, self, self.earth

def Status(self):
    return self.status

# set up
initialize()
register(SuperBeing, name="SuperBeing")  # register SuperBeing class with name
Earth = Resource(2, name="Earth")  # set name to ensure window title is set
register(Earth)  # register Earth Resource
SB = SuperBeing(Earth)
activate(SB, SB.Create())  # when activated SB will be registered
```
- **Line 2**: Import `SimulationGUIDebug` here.
- **Lines 21, 43**: Here the `name` variable is set so that the registered windows may use it as the title.
- **Line 15**: Here the instance `man` is registered with the user hook as `man.Status` and window is created for the process.
- **Line 41**: Here the SuperBeing class (NOT an instance) is registered. So now whenever a SuperBeing instance is activated SimulationGUIDebug will automatically register it using the `name` parameter `SuperBeing` as the window’s title.
- **Line 44**: Here the resource Earth is registered and a window is created for the resource.
- **Line 47**: Note that SB was not registered here. That is because its class was already registered on line 40. Once SB is activated, it will automatically be registered.

![Figure 5: Example.py Preview](image)
6 SimulationGUIDebug Backend

6.1 Backend Goal

When creating the Debugging backend, one of our primary goals was to make the interface for the user similar to other familiar debuggers and require as little change to the program as possible. In order to use the debugger, the program just needs to include SimulationGUIDebug, instead of SimPy, by using `from SimulationGUIDebug import *`. Otherwise, all function calls to our debugger are the same as they would be in SimPy. Also, the interactive command-line interface works very much like PDB or DDD does, with basic commands performing the debugging.

6.2 Backend Implementation

Our actual debugger works through SimPy’s callback functionality. Instead of letting the user specify a callback function to SimPy, we use our own `callbackFunction()` function. Since SimPy calls this callback function after every event has fired, our `callbackFunction()` simply needs to see if there is a breakpoint specified for the event’s time and, if so, perform our GUI updates. Since SimPy’s callback functionality is blocking, so that the simulation is suspended while the callback function is executing, the simulation is truly stopping when a breakpoint is hit. Since we are using SimPy’s callback function ourselves, we implement our own callback functionality to allow the user to still use this feature.

To get the interaction with the user, there is a function `promptUser()`, which uses Python’s `raw_input()` function to get the commands from the user. The basic debugging functions exist: `[s]` to step to the next event in the simulation, `[c]` to continue to the next breakpoint, `[b #]` to add a breakpoint (tuples are allowed, in order to add multiple breakpoints), `[q]` to quit the simulation. We also save the last command that was issued, so entering no command will run the previously executed command, much like other debuggers.

In addition to our debugger, we wanted users to be able to utilize existing debuggers to work with their programs. In order to achieve this goal, we added a feature which allows the user to specify the run mode, via the function `setRunMode()`. The default value is `STEP`, which runs the debugger as expected. The alternate value is `NO_STEP`, which will not evaluate breakpoints at all, but continues to update the GUI. This way, a user can specify breakpoints and step through the program in an external debugger such as PDB, but still utilize our GUI for debugging purposes.

7 GUIDebug.py (GUI Implementation)

GUIDebug is the GUI package that SimulationGUIDebug uses to create its GUI. Users who wish to use SimulationGUIDebug do not to know the details of GUIDebug. The following information is useful for users who wish to expand the GUI capabilities of SimulationGUIDebug.

7.1 GUIController Class

7.1.1 Use

The GUIController is used to control all of the GUI windows that are currently open, and to create new windows. It currently contains methods to add a new GenericWindow, ProcessWindow, and ResourceWindow
as well as to update the GUI of all currently opened windows.

7.1.2 GUIController API

addNewWindow

- **Call:** `addNewWindow(self, obj, name, hook)`
- **Description:** Creates and adds a new GenericWindow object to the controller’s window list.
- **Parameters:** `obj` is the object to be associated with the window. `hook` is a function that returns a string that will be printed to the window.
- **Return Value:** None

addNewProcess

- **Call:** `addNewProcess(self, obj, name, hook)`
- **Description:** Creates and adds a new ProcessWindow object to the controller’s process window list.
- **Parameters:** `obj` is the object to be associated with the window. `hook` is a function that returns a string that will be printed to the window.
- **Return Value:** None

addNewResource

- **Call:** `addNewResource(self, obj, name, hook)`
- **Description:** Creates and adds a new ResourceWindow object to the controller’s resource window list.
- **Parameters:** `obj` is the object to be associated with the window. `hook` is a function that returns a string that will be printed to the window.
- **Return Value:** None

updateAllWindows

- **Call:** `updateAllWindows(self)`
- **Description:** Calls update() for each window in the controller’s window list. Then calls organizeWindows() to clean up and organize the current windows on the screen.
- **Parameters:** None
- **Return Value:** None

saveNextEvent

- **Call:** `saveNextEvent(self)`
- **Description:** Saves the next event that will be run in self.nextEvent.
- **Parameters:** None
- **Return Value:** None

**removeWindow**

- **Call:** `removeWindow(w)`
- **Description:** Removes all instances of w in GUIController’s window lists.
- **Parameters:** `w`: GenericWindow to be removed from controller.
- **Return Value:** None

**organizeWindows**

- **Call:** `organizeWindows(self)`
- **Description:** Organizes the windows so that the event list window is in the top left corner of the screen, the process windows are to the right of the event list window, and the resource windows are below the event list window.
- **Parameters:** None
- **Return Value:** None

## 8 Source Code

### 8.1 SimulationGUIDebug.py

```python
from SimPy.SimulationStep import *
from Tkinter import *
import SimPy.SimulationStep, GUIDebug

# global variables
_breakpoints = []
_until = 0
_callback = None
_lastCommandIssued = ""
_simStarted = False
_registeredClasses = []
_runMode = None

# run modes
STEP = 1
NO_STEP = 2

# register new object for windowing
def register(obj,hook=lambda :",name=None):
    global _registeredClasses
    # if process subclass is given register it
    if type(obj) == TypeType and issubclass(obj, Process):
        _registeredClasses += [(obj,name,hook)]
```

# if instance of process is given register it
elif issubclass(type(obj), Process):
    _guiCtrl.addNewProcess(obj, name, hook)

# if instance of Resource is given register it
elif issubclass(type(obj), Resource):
    _guiCtrl.addNewResource(obj, name, hook)

# else create a generic window with hook
else:
    _guiCtrl.addNewWindow(obj, name, hook)

# override activate to catch registered class instances
def activate(obj, process, at="undefined", delay="undefined", prior=False):
    global _registeredClasses
    SimPy.SimulationStep.activate(obj, process, at, delay, prior)
    # if obj is instance of the class register it
    for c, n, h in _registeredClasses:
        if isinstance(obj, c):
            _guiCtrl.addNewProcess(obj, n, h)

# add to breakpoints
def newBreakpoint(newBpt):
    global _breakpoints
    _breakpoints.append(newBpt)
    _breakpoints.sort()

# set the current run mode of simulation
def setRunMode(runMode):
    global _runMode
    _runMode = runMode

# initialize the simulation and the GUI
def initialize():
    SimPy.SimulationStep.initialize()
    # create gui controller
    global _guiCtrl
    _guiCtrl = GUIDebug.GUIController()
    # initialize run mode if not already set
    global _runMode
    if not _runMode:
        _runMode = STEP

# simulation function
def simulate(callback=lambda : None, until=0):
    global _runMode
    # print usage
    if(_runMode == STEP):
        print "Breakpoint Usage:"
        print "  [c] Continue simulation"
        print "  [s] Step to next event"
        print "  [b #] Add new breakpoint"
        print "  [q] Quit debugger"

    # set global variables
global _until
_until = until

global _callback
_callback = callback

# initialize to step command
global _lastCommandIssued
_lastCommandIssued = "s"

# only prompt user if we are in STEP mode
if( _runMode == STEP): promptUser()

# quit if user entered 'q'
if( _lastCommandIssued == 'q'): return

# begin simulation
if( _runMode == NO_STEP)
    _guiCtrl.updateAllWindows()
    return

if( 0 == len(_breakpoints) ): return

# this is a breakpoint
if( now() >= _breakpoints[0] ):
    # update gui
    _guiCtrl.updateAllWindows()

    # remove past times from breakpoints list
    while( 0 != len(_breakpoints) and now() >= _breakpoints[0] ):
        _breakpoints.pop(0)

    # call user's callback function
    global _callback
    _callback()

    promptUser()

# prompt user for next command
def promptUser():

    global _simStarted

    # set prompt text
    prompt = '(SimDB) > ' + SimPy.SimulationStep.simulate(callback=callbackFunction, until=_until)

    # pause for breakpoint
    while( 1 ):
        input = raw_input( prompt )

        # take a look at the last command issued
        global _lastCommandIssued
if 0 == len(input):
    input = _lastCommandIssued

_lastCommandIssued = input

# continue
if( "c" == input ):
    break

# step
elif( "s" == input ):
    global _breakpoints
    _breakpoints.insert(0,0)
    break

# add breakpoint
elif( 0 == input.find("b")): try:
    for i in eval( input[1:] + "," ):
        newBreakpoint( int(i) )
except SyntaxError:
    print "missing breakpoint values"

# quit
elif( "q" == input ):
    SimPy.SimulationStep.stopSimulation()
    return

else:
    print " unknown command"

8.2 GUIDebug.py

from Tkinter import *
from SimPy.SimulationStep import now,Globals

# Creates and controls the GUI of the program
class GUIController(object):

def __init__(self):
    self.root = Tk()
    self.root.withdraw()

    self.saveNextEvent()

    self.eventWin = EventWindow(self)
    self.wlist = []
    self.plist = []
    self.rlist = []

    # Adds a new Window to the GUI
def addNewWindow(self, obj, name, hook):
        self.wlist += [GenericWindow(obj, hook, self, name)]

    # Adds a new Process to the GUI
def addNewProcess(self, obj, name, hook):
        self.plist += [ProcessWindow(obj, hook, self, name)]

    # Adds a new Resource to the GUI
def addNewResource(self, obj, name, hook):
        self.rlist += [ResourceWindow(obj, hook, self, name)]

    # Updates all the windows currently up
def updateAllWindows(self):
    for w in self.wlist: w.update()
for p in self.plist: p.update()
for r in self.rlist: r.update()
if self.eventWin.window: self.eventWin.update()
self.organizeWindows()
self.saveNextEvent()

# removes all instances of window in lists
def removeWindow(self, w):
f = lambda win: win is not w
self.wlist = filter(f, self.wlist)
self.plist = filter(f, self.plist)
self.rlist = filter(f, self.rlist)

# save next event to be run
def saveNextEvent(self):
tempList = []
tempList[:] = Globals.sim.e.timestamps
tempList.sort()
for ev in tempList:
    # return only event notices which are not cancelled
    if ev[3]: continue
    # save next event
    self.nextEvent = ev
    return
self.nextEvent = (None, None, None, None)

def organizeWindows(self):
    # event window
    eventWindowHeight = 0
    # only organize event window only if it exists
    if self.eventWin.window:
        eventWindowHeight = 40 + 20 * self.eventWin.table.size()
        self.eventWin.setSize(500, eventWindowHeight, 20, 40)
    # generic windows
    count = -1
    for win in self.wlist:
        count += 1
        (w, h, x, y) = win.getWindowSize()
        win.setSize(w, h, 20, 40 + eventWindowHeight + 40)
        eventWindowHeight += h + 40
    # process windows
    xCount = -1
    yCount = 0
    for p in self.plist:
        xCount += 1
        yCoord = 40 + 150 * xCount
        xCoord = 550 + 210 * yCount
        p.setSize(200, 120, xCoord, yCoord)
        if yCoord >= 600:
xCount = -1
yCount += 1

# resource windows
count = -1
for r in self.rlist:
count += 1

windowHeight = 0
windowHeight += 20  # capacity title
windowHeight += 105  # empty table sizes
windowHeight += (r.activeT.size() + r.waitT.size()) * 17  # add size for each row
r.setWindowSize(200, windowHeight, 20 + 220 * count, 40 + eventWindowHeight + 40)

# Creates a basic window that shows a user made hook.
class GenericWindow(object):
    def __init__(self, obj, hook, guiCtrl, title=None):
        self.window = Toplevel()
        self.window.protocol("WM_DELETE_WINDOW", self._destroyWindow)
        self.obj = obj
        self.hook = hook
        self.guiCtrl = guiCtrl
        if not title:
            self.title = "%s%s" % (type(obj), id(obj))
        else:
            self.title = title
        self.initGUI()

    def setWindowSize(self, w, h, x, y):
        newG = "%dx%d+%d+%d" % (w, h, x, y)
        self.window.geometry(newG)

    def setWindowOrigin(self, x, y):
        (w, h, xx, yy) = self.getWindowSize()
        newG = "%dx%d+%d+%d" % (w, h, x, y)
        self.window.geometry(newG)

    def getWindowSize(self):
        g = self.window.geometry()
        return [int(i) for i in g.replace('+', 'x').split('x')]

    def _destroyWindow(self):
        self.window.destroy()
        self.window = None
        self.guiCtrl.removeWindow(self)

    # Creates the window
    def initGUI(self):
        self.window.title(self.title)
txt = self.hook()
if txt != "":
txt += '\n'
self.hookTxt = Label(self.window, text=txt, justify=LEFT)
self.hookTxt.pack()

    # Updates the window
    def update(self):
        txt = self.hook()
if txt != "":
txt += '\n'
self.hookTxt['text'] = txt

    # Class that creates the event window for the simulation that
# displays the time and event.
class EventWindow(GenericWindow):

def __init__(self, guiCtrl):
    self.window = Toplevel()
    self.window.protocol("WM_DELETE_WINDOW", self._destroyWindow)
    self.guiCtrl = guiCtrl
    self.initGUI()

# Creates the initial window using a two column window with a
# status bar on the bottom

def initGUI(self):
    self.window.title("Event List")
    # Creates the table
    self.table = MultiListbox(self.window, (('', 1), ('Time', 15),
                                              ('Process', 20), ('Next Line', 5)))
    # Adds the status bar to display the current simulation time
    self.status = StatusBar(self.window)
    self.status.pack(side=TOP, fill=X)

    self.update()

# Updates the window
def update(self):
    self.updateETable()
    self.updateStatus()

# Updates the status bar

def updateStatus(self):
    self.status.set(" Current Time: %s", now())

# Updates the table

def updateETable(self):
    self.table.delete(0, self.table.size())
    tempList = []
    tempList.sort()
    tempList[:] = Globals.sim._e.timestamps
    ev = self.guiCtrl.nextEvent
    nextLine = 0
    if ev[2]:
        if ev[2]._nextpoint:
            nextLine = ev[2]._nextpoint.gi_frame.f_lineno

    if not ev[3]:
        continue

    count = -1
    currentEvent = ''

    for ev in tempList:
        # return only event notices which are not cancelled
        if ev[3]: continue

        count += 1
        currentEvent = ''

        #if count == 0 and now() == ev[0]:
        #    currentEvent = ''

        nextLine = 0
        if ev[2]:
            if ev[2]._nextpoint:
                nextLine = ev[2]._nextpoint.gi_frame.f_lineno

            self.table.insert(END, (' >>',
                                      str(ev[3]), ev[2].name, nextLine ))

        self.table.insert(END, (currentEvent,
                                      str(ev[3]), ev[2].name, nextLine ))
str(ev[0]), ev[2].name, nextLine )

self.table.pack(expand=YES, fill=BOTH)

# Creates a Process Window that shows the status, Next Event time, # if the Process is currently interrupted, and an optional user hook.
class ProcessWindow(GenericWindow):

def __init__(self, obj, hook, guiCtrl, name):
    self.proc = obj
    if name:
        obj.name = name
    GenericWindow.__init__(self, obj, hook, guiCtrl, "Process: %s" % obj.name)

    # Initializes the window
    def initGUI(self):
        # Creates the table
        self.table = MultiListbox(self.window, ((None, 10), (None, 15)))
        self.status = StatusBar(self.window)
        self.status.pack(side=BOTTOM, fill=X)
        GenericWindow.initGUI(self)
        self.setWindowSize(0, 0, -1000, -1000)

        # Updates the window
        def update(self):
            # If the process has been terminated close the window
            if self.proc.terminated():
                self._destroyWindow()
                return

            if self.isRunning():
                self.status.label["text"] = "Running!"
                self.status.label["fg"] = "green"
            else:
                self.status.label["text"] = ""  
                self.status.label["fg"] = "white"

            self.table.delete(0, self.table.size())

            if self.proc.active() == False:
                status = "Passive"
            else:
                status = "Active"

            if self.proc._nextTime:
                nextEvent = self.proc._nextTime
            else:
                nextEvent = ""

            if self.proc.interrupted() == True:
                interrupted = "True"
            else:
                interrupted = "False"

            self.table.insert(END, (" Status:" , status))
            self.table.insert(END, (" Next Event:" , nextEvent))
            self.table.insert(END, (" Interrupted:" , interrupted ))

            self.table.pack(expand=YES, fill=BOTH)
            GenericWindow.update(self)

        def isRunning(self):

def main():
    app = Application()
    app.master.title("Process Management")
    app.mainloop()
return self.guiCtrl.nextEvent[2] is self.proc

# Creates a Resource Window that displays the capacity, waitQ, activeQ and an optional user hook
class ResourceWindow(GenericWindow):
    def __init__(self, obj, hook, guiCtrl, name):
        self.resource = obj
        if name:
            obj.name = name
        GenericWindow.__init__(self, obj, hook, guiCtrl, "Resource: %s" % obj.name)

    # Initializes the window with the two tables for the waitQ and activeQ
    def initGUI(self):
        Label(self.window, text="Capacity: %d" % self.resource.capacity).pack()

        self.activeT = MultiListbox(self.window, (('#', 5), ('ActiveQ', 20)))
        self.waitT = MultiListbox(self.window, (('#', 5), ('WaitQ', 20)))
        self.updateQTables()

        GenericWindow.initGUI(self)
        self.setWindowSize(0, 0, -1000, -1000)

    def update(self):
        GenericWindow.update(self)
        self.updateQTables()

    # Updates the waitQ and activeQ tables
    def updateQTables(self):
        self.activeT.delete(0, END)
        self.waitT.delete(0, END)

        self.activeT.pack(expand=YES, fill=BOTH)
        self.waitT.pack(expand=YES, fill=BOTH)

        # A class that creates a multilistbox with a scrollbar
        class MultiListbox(Frame):
            def __init__(self, master, lists):
                Frame.__init__(self, master)
                self.lists = []

                for l, w in lists:
                    frame = Frame(self); frame.pack(side=LEFT, expand=YES, fill=BOTH)

                    if l is None:
                        None
                    elif l is '':
                        Label(frame, text='', borderwidth=1, relief=FLAT).pack(fill=X)
                    else:
                        Label(frame, text=l, borderwidth=1, relief=SOLID).pack(fill=X)

                    lb = Listbox(frame, width=w, height=0, borderwidth=0, selectborderwidth=0,
                                 relief=FLAT, exportselection=FALSE)
                    lb.pack(expand=YES, fill=BOTH)
                    self.lists.append(lb)

                    frame = Frame(self); frame.pack(side=LEFT, fill=Y)
                    Label(frame, borderwidth=1, relief=RAISED).pack(fill=X)
sb = Scrollbar(frame, orient=VERTICAL, command=self._scroll)
sb.pack(expand=YES, fill=Y)
self.lists[0]['yscrollcommand'] = sb.set

def _select(self, y):
    row = self.lists[0].nearest(y)
    self.selection_clear(0, END)
    self.selection_set(row)
    return 'break'

def _button2(self, x, y):
    for l in self.lists:
        l.scan_mark(x, y)
    return 'break'

def _b2motion(self, x, y):
    for l in self.lists:
        l.scan_dragto(x, y)
    return 'break'

def _scroll(self, *args):
    for l in self.lists:
        apply(l.yview, args)

def curselection(self):
    return self.lists[0].curselection()

def delete(self, first, last=None):
    for l in self.lists:
        l.delete(first, last)

def get(self, first, last=None):
    result = []
    for l in self.lists:
        result.append(l.get(first, last))
    if last:
        return apply(map, [None] + result)
    return result

def index(self, index):
    self.lists[0].index(index)

def insert(self, index, *elements):
    for e in elements:
        i = 0
        for l in self.lists:
            l.insert(index, e[i])
            i = i + 1

def size(self):
    return self.lists[0].size()

def see(self, index):
    for l in self.lists:
        l.see(index)

def selection_anchor(self, index):
    for l in self.lists:
        l.selection_anchor(index)

def selection_clear(self, first, last=None):
    for l in self.lists:
        l.selection_clear(first, last)

def selection_includes(self, index):
    return self.lists[0].selection_includes(index)

def selection_set(self, first, last=None):
    for l in self.lists:
        l.selection_set(first, last)
# Creates a statusbar

class StatusBar(Frame):

    def __init__(self, master):
        Frame.__init__(self, master)
        self.label = Label(self, bd=1, relief=SUNKEN, anchor=W)
        self.label.pack(fill=X)

    def set(self, format, *args):
        self.label.config(text=format % args)
        self.label.update_idletasks()

    def clear(self):
        self.label.config(text="")
        self.label.update_idletasks()