Abstract

Describes the operation and implementation of a graphical user interface to be used by administrators of our Course Management System to create and maintain the file defining the current set of courses and sections.

This publication is available in Web form and also as a PDF document. Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Workflow for CMS course and section creation ................................................................. 7
2. Glossary .............................................................................................................................. 8
   2.1. Section ....................................................................................................................... 8
   2.2. Course ..................................................................................................................... 8
   2.3. Metacourse ............................................................................................................... 8
   2.4. Cross-listed ............................................................................................................. 9
3. Design notes ....................................................................................................................... 9
   3.1. Required output ...................................................................................................... 9
   3.2. User stories ............................................................................................................. 10
   3.3. Inputs ...................................................................................................................... 12
   3.4. General workflow .................................................................................................. 12
   3.5. Design of the UID finder ...................................................................................... 13
   3.6. The archive subdirectories: Accountability considerations for uploading ............... 14
4. General layout ................................................................................................................... 15
   4.1. Layout of the general controls panel ................................................................. 16
   4.2. Layout of the primary instructor panel ............................................................... 17
   4.3. Layout of the department schedule panel ......................................................... 18
   4.4. Layout of the section edit panel ........................................................................... 18
   4.5. Layout of the metacourse panel .......................................................................... 19
5. Operation .......................................................................................................................... 19
6. Object architecture .......................................................................................................... 21
   6.1. Control flow .......................................................................................................... 22
   6.2. Requests originating from App ........................................................................... 22
   6.3. Requests originating from ReqPanel .................................................................... 23
   6.4. Requests originating from SchedPanel .............................................................. 23
   6.5. Requests originating from MetaPanel ................................................................. 23
   6.6. Requests originating from SectionPanel ............................................................. 23
7. Code prologue .................................................................................................................. 24

1 http://www.nmt.edu/tcc/projects/cmsadds/
2 http://www.nmt.edu/tcc/projects/cmsadds/cmsadds.pdf
8. Module imports ..................................................................................................................... 24
9. Manifest constants .................................................................................................................. 25
  9.1. PROGRAM_NAME ............................................................................................................... 25
  9.2. VERSION .......................................................................................................................... 25
  9.3. SCHEMA_FILE .................................................................................................................. 26
  9.4. SCHED_FILE_PATH ......................................................................................................... 26
  9.5. ADDS_PREFIX ..................................................................................................................... 26
  9.6. XML_EXT .......................................................................................................................... 26
  9.7. ARCH_PREFIX .................................................................................................................... 26
  9.8. TIME_TAIL_FORMAT ....................................................................................................... 26
  9.9. ADVISING_EMAIL .......................................................................................................... 26
  9.10. CMSACCTS_CMD ........................................................................................................... 27
  9.11. CMSSECTS_CMD ............................................................................................................ 27
  9.12. CMSENROLL_CMD ......................................................................................................... 27
  9.13. MOOROSTER_CMD ......................................................................................................... 27
  9.14. OUT_FILE_LINE_PAT .................................................................................................... 27
  9.15. MOODLE_UPLOAD_CMD ................................................................................................. 28
  9.16. NEXT_SUMMER_DATE ................................................................................................. 28
  9.17. NEXT_SPRING_DATE ..................................................................................................... 28
  9.18. SPRING_SUMMER_DATE ............................................................................................... 28
  9.19. SUMMER_FALL_DATE ................................................................................................. 28
  9.20. FALL_SPRING_DATE ...................................................................................................... 28
  9.21. BASE_FONT_FAMILY .................................................................................................... 29
  9.22. NARROW_FONT_FAMILY ............................................................................................... 29
  9.23. MONOSPACED_FONT_FAMILY ..................................................................................... 29
  9.24. ENTRY_STYLE ................................................................................................................ 29
  9.25. FRAME_STYLE ............................................................................................................... 29
  9.26. LABEL_FRAME_STYLE .................................................................................................. 29
  9.27. PANEL_TITLE_STYLE .................................................................................................. 29
  9.28. MAIN_BG_COLOR .......................................................................................................... 29
  9.29. MAIN_FG_COLOR .......................................................................................................... 30
  9.30. ACTIVE_BG_COLOR ........................................................................................................ 30
  9.31. REQ_WIDE .................................................................................................................... 30
  9.32. REQ_HIGH ..................................................................................................................... 30
  9.33. SWAP_WIDE .................................................................................................................. 30
  9.34. SWAP_HIGH .................................................................................................................. 30
  9.35. GUI text constants ........................................................................................................ 30
  9.36. Popup text constants ...................................................................................................... 30
10. Specification functions ........................................................................................................ 38
  10.1. adds-file-name: What is the name of the XML file? ................................................... 38
  10.2. arch-dir-name: Archive directory name ...................................................................... 39
  10.3. arch-file: Name of an archival backup file ................................................................. 39
  10.4. cal-year: Academic year to calendar year .................................................................. 39
  10.5. default-cal-year: Default calendar year for a given semester .................................. 39
  10.6. default-cal-year-now: Default calendar year with no semester given .................. 40
  10.7. default-semester-now: Default semester ................................................................. 40
  10.8. prefix-match-count: Number of matching strings ...................................................... 40
  10.9. semester-name: Semester code to semester name ..................................................... 40
  10.10. semester-picker-callback: Callback for change of semester .................................... 41
  10.11. std-sem-code: Standardize an abbreviated semester code ...................................... 41
  10.12. transient-can-close: Can the transient panel be removed? ..................................... 41
  10.13. Blanket preconditions ................................................................................................. 41
11. Main program: `main()` ................................................................................................................................. 42
12. class `Args()`: Command line argument singleton .......................................................................................... 43
   12.1. `Args.__init__()........................................................................................................................................... 44
   12.2. `Args._syntaxCheck()`: Syntax-check the command line ............................................................................. 45
   12.3. `Args._checkSemester()`: Validate the semester name .............................................................................. 46
   12.4. `Args._checkYear()`: Validate the calendar year ....................................................................................... 47
   12.5. `Args._applyDefaults()`: Set defaults for year and semester code ............................................................. 47
13. Locally defined exceptions ..................................................................................................................................... 48
14. class `App`: The application as a whole ............................................................................................................... 48
   14.1. `App.semSched`: Current semester schedule .............................................................................................. 53
   14.2. `App._init__().............................................................................................................................................. 53
   14.3. `App._styler()`: Set up fonts and `ttk` styles ............................................................................................... 54
   14.4. `App.createWidgets()` .................................................................................................................................. 55
   14.5. `App._selInstrHandler()`: Bring up one instructor's course list ................................................................. 56
   14.6. `App._newTeacher()`: Create a new `readadds.Teacher` from a `uidfinder.TccAccount` .......... 58
   14.7. `App.mainWidgets()`: Create the widgets in the general controls panel ......................................................... 59
   14.8. `App._createControls()`: Create general controls .................................................................................... 59
   14.10. `App._saveHandler()`: The `Save adds-season-yyyy.xml` button ................................................................. 62
   14.11. `App.uploadAcctsHandler()`: The `Upload cms accounts` button .............................................................. 62
   14.12. `App._imsUpload()`: Build and upload an IMS file into the CMS ............................................................... 63
   14.13. `App.runImsCommand()`: Build an IMS file .................................................................................................. 64
   14.15. `App._copyCms()`: Transfer an IMS Enterprise file to the CMS server ......................................................... 66
   14.16. `App.uploadCoursesHandler()`: The `Upload cms courses` button ............................................................ 67
   14.17. `App.uploadEnrollHandler()`: The `Upload cms enrollments` button ............................................................ 68
   14.18. `App.cleanupDropsHandler()`: The `Cleanup cms drops` button ................................................................. 69
   14.19. `App.saveOrCancel()`: Insure changes have been saved ............................................................................... 70
   14.20. `App.quitHandler()`: The `Quit` button ........................................................................................................ 71
   14.21. `App.reqWidgets()`: Create widgets in the primary instructor panel ............................................................. 72
   14.22. `App.swapWidgets()`: Create the swap area and its children ........................................................................ 73
   14.23. `App.addSectHandler()`: Add a section ......................................................................................................... 74
   14.24. `App.lookupDeptName()`: Look up a department code ................................................................................. 75
   14.25. `App.lookupCRN()`: Find a CRN in the current semester ............................................................................. 75
   14.26. `App.lookupCRNSched()`: Find a scheduled section by CRN ......................................................................... 75
   14.27. `App.close()`: Remove the current semester ............................................................................................... 75
   14.28. `App.saveOrDiscard()`: Ask the user if they want to save their changes ........................................................ 77
   14.29. `App.closeTransient`: Clear the swap area .................................................................................................. 78
   14.30. `App.openSemester()`: Switch to a different semester .................................................................................. 78
   14.31. `App.showSemester()`: Change semester-dependent widgets ........................................................................ 79
   14.32. `App.saver()`: Write back the XML ............................................................................................................. 80
   14.33. `App.archFileName()`: Archive directory name ............................................................................................ 81
   14.34. `App.archDirName()`: Generate a name for the archive file ........................................................................... 82
   14.35. `App.makeArchDir()`: Build the archive directory if necessary .......................................................................... 82
   14.36. `App.reqBlank()`: Blank out the `ReqPanel` .................................................................................................. 83
   14.37. `App.reqSelect()`: Display a selected instructor .......................................................................................... 83
   14.38. `App.removeInstructor()`: Delete a primary instructor's records ...................................................................... 84
   14.39. `App.showDept()`: Display the schedule for one department ........................................................................ 84
   14.40. `App.showSection()`: Bring up the `SectionPanel` ........................................................................................... 85
   14.41. `App.showMeta()`: Bring up the `MetaPanel` ................................................................................................ 86
   14.42. `App.deleteMeta()`: Remove a metacourse .................................................................................................. 87

New Mexico Tech Computer Center
cmsadds: A GUI for CMS course creation
14.43. App.getSelectedSection(): What section is selected in the ReqPanel? ............... 87
14.44. App.clearSpecificMeta() ...................................................................................... 87
14.45. App.updateReqPanel(): Handler for a change in a metacourse name .................... 88
15. class SemesterPicker: Widget to show and change the current semester ..................... 88
15.1. SemesterPicker.set(): Change the displayed semester ............................................ 89
15.2. SemesterPicker.__init__() ......................................................................................... 90
15.3. SemesterPicker._createWidgets() .......................................................................... 90
15.4. SemesterPicker._addSemester(): Add one semester command to the semester menu ... 91
16. class ReqPanel: The primary instructor edit panel ..................................................... 92
16.1. ReqPanel.select(): Display a given instructor .......................................................... 95
16.2. ReqPanel.addSection(): Add a section to the current instructor .............................. 96
16.3. ReqPanel.getSelSection(): Which section is selected? .......................................... 97
16.4. ReqPanel.close(): Remove this panel ................................................................... 98
16.5. ReqPanel.updateLists(): Update the section and metacourse lists .......................... 98
16.6. ReqPanel.deleteMeta(): Remove a metacourse ...................................................... 99
16.7. ReqPanel._init__() .................................................................................................. 99
16.8. ReqPanel._createWidgets() .................................................................................... 100
16.9. ReqPanel._createEditMetaHandler(): Create the name-editing widgets ................. 101
16.10. ReqPanel._updateNames(): Save the contents of the name fields ....................... 103
16.11. ReqPanel._createControlFrame(): Add panel controls ........................................ 103
16.12. ReqPanel._removeInstrHandler(): Handler for Remove this instructor from the CMS 104
16.13. ReqPanel._createSectionsFrame(): Create the section list area ............................ 106
16.14. ReqPanel._sectionsListHandler(): Handler for clicking on a section in the sections list ........................................................................................................ 107
16.15. ReqPanel._crnHandler(): User entered a CRN to be added .................................... 108
16.16. ReqPanel._delSelSectionHandler: Handler for the Delete selected section button ............................................................................................................. 109
16.17. ReqPanel._deptHandler(): Show a department's schedule in the swap area .......... 110
16.18. ReqPanel._createMetaFrame(): Create the metacourse widget group .................... 111
16.19. ReqPanel._addMetaHandler(): Handler for the Add metacourse button ............... 112
16.20. ReqPanel._delSelMetaHandler(): Handler for the Delete selected metacourse button ............................................................................................................. 113
16.21. ReqPanel._editMetaHandler(): Bring up a metacourse for editing ....................... 114
17. class ScrolledList: Generic listbox with one or two scrollbars ................................... 115
18. class Pager: Base widget with swappable content ..................................................... 116
18.1. Pager.add(): Register a child frame ....................................................................... 118
18.2. Pager.select(): Make a child appear ................................................................. 118
18.3. Pager.clear(): Erase the window ........................................................................ 118
18.4. Pager.get(): Return the currently displayed widget .......................................... 119
18.5. Pager._init__() ...................................................................................................... 119
19. class SectionsList: A scrollable listbox of section lines ........................................... 119
19.1. SectionsList._init__() ............................................................................................. 120
19.2. SectionsList.clear() .............................................................................................. 121
19.3. SectionsList.add(): Add a new section ............................................................. 122
19.4. SectionsList.rebuild(): Rebuild the displayed list ............................................. 122
19.5. SectionsList._showLine(): Format a line displaying a SectionSchedule ............ 122
19.6. SectionsList.genSections(): Generate the sections ............................................. 122
19.7. SectionsList.delete() ......................................................................................... 123
19.8. SectionsList.get(): Retrieve the currently selected SectionSchedule ................ 123
19.9. SectionsList.nSections(): How many child sections are there? ............................. 124
19.10. SectionsList._clickHandler(): Handler for clicking on an item ........................................... 124
20. class DeptSelector: Widget to bring up a department's schedule .............................................. 125
20.1. DeptSelector.select(): Display a given department code and name ..................................... 126
20.2. DeptSelector._init__(): ........................................................................................................ 126
20.3. DeptSelector._buildList(): Build the list of departments ....................................................... 127
20.4. DeptSelector._byDeptName(): Sort departments by name ..................................................... 128
20.5. DeptSelector._createWidgets() ........................................................................................... 128
20.6. DeptSelector._addDept(): Add one department command .................................................. 129
20.7. DeptSelector._findDeptHandler(): Handler to look up a department code .......................... 130
20.8. DeptSelector._show(): Display the schedule for the current department .............................. 131
21. class MetaList: Listbox showing the instructor's metacourses .................................................... 131
21.1. MetaList.add(): Add a cross-listed set .................................................................................. 132
21.2. MetaList._rebuild(): Rebuilt the listbox .............................................................................. 132
21.3. MetaList.delete(): Remove a cross-listed set ....................................................................... 133
21.4. MetaList.clear(): Clear the list ............................................................................................. 133
21.5. MetaList.getSelected(): Which metacourse is selected? .................................................... 134
21.6. MetaList._init__(): ............................................................................................................. 134
21.7. MetaList._clickHandler(): Handle right-click on the metacourse list .................................... 134
22. class SwapArea: Home for transient panels .............................................................................. 135
22.1. SwapArea._init__(): ............................................................................................................. 136
22.2. SwapArea.close(): Blank the swap area ................................................................................. 136
23. class Transient(): Base class for panels in the swap area .......................................................... 136
23.1. Transient._init__(): ............................................................................................................ 137
23.2. Transient.close(): Default close function ............................................................................. 137
24. class SchedPanel: Display all the sections in a given department .................................................. 138
24.1. SchedPanel._init__(): ........................................................................................................... 139
24.2. SchedPanel._createWidgets() .................................................................................................. 139
24.3. SchedPanel._addHandler(): Handler for Add selected section ............................................. 140
24.4. SchedPanel.select(): Change the displayed schedule ............................................................ 141
24.5. SchedPanel._addSect(): Add one section line to the listbox .................................................... 141
25. class SectionPanel: Edit section information .......................................................................... 142
25.1. SectionPanel._init__(): ....................................................................................................... 145
25.2. SectionPanel._createWidgets() ............................................................................................ 145
25.3. SectionPanel._createTitle(): Create panel title widgets .................................................... 146
25.4. SectionPanel._closeHandler: Handler for the Close button ................................................. 147
25.5. SectionPanel._createDesc(): Create description editing widgets .......................................... 148
25.6. SectionPanel._descValidate(): Validate the section title ....................................................... 149
25.7. SectionPanel._createReload(): Create reload editing widgets .............................................. 150
25.8. SectionPanel._reloadValidate(): Update the reload attribute .................................................. 150
25.9. SectionPanel._createRolesFrame() ....................................................................................... 151
25.10. SectionPanel._createRolesGroup(): Create the role requests list ........................................ 152
25.11. SectionPanel._delSelRoleHandler: Handler for the Delete selected role button .................. 153
25.12. SectionPanel._advisingCBHandler: Handler for the Advising Center access checkbutton ................................................................................................................................. 154
25.13. SectionPanel._createAddGroup(): Create the role add buttons ............................................ 155
25.14. SectionPanel._makeRoleButton(): Create one add-role button ........................................... 156
25.15. SectionPanel._roleHandler(): Generic handler for an add-role button ............................. 157
25.16. SectionPanel.select(): Display a readadds .Section instance ............................................. 158
25.17. SectionPanel._rolesUpdate(): Rebuild the displays of roles ................................................ 159
25.18. SectionPanel.showRole(): Display one role request ............................................................ 160
25.19. SectionPanel.close() ........................................................................................................... 161
26. class MetaPanel: Edit cross-listed sets (metacourses) ........................................... 162
   26.1. MetaPanel.select(): Display a metacourse ....................................................... 164
   26.2. MetaPanel.__init__() ......................................................................................... 165
   26.3. MetaPanel.createWidgets() ............................................................................... 165
   26.4. MetaPanel._createTitle(): Create the panel title and close button ................. 166
   26.5. MetaPanel._closeHandler(): Handler for the Close button ......................... 167
   26.6. MetaPanel._createDescFrame(): Create title and reload fields ..................... 167
   26.7. MetaPanel._descValidate(): Validate the metacourse description field ......... 168
   26.8. MetaPanel._updateReload(): Update the reload field ................................ 169
   26.9. MetaPanel._createControlFrame(): Create panel controls ......................... 169
   26.10. MetaPanel._addSectHandler(): Handler for the Add selected section button .... 170
   26.11. MetaPanel._delSectHandler(): Handler for the Delete selected section button ... 171
   26.12. MetaPanel.clear(): Clear out the content ....................................................... 171
   26.13. MetaPanel.close() ......................................................................................... 172
27. class MultiDialog: Dialog popup .......................................................................... 173
   27.1. MultiDialog.__init__() ......................................................................................... 174
   27.2. MultiDialog.body(): Place widgets in the popup ............................................ 174
   27.3. MultiDialog.buttonbox(): Create the choice buttons ...................................... 174
28. addsFileName(): The name of the save file ......................................................... 175
29. message(): Write a message to stderr ................................................................. 176
30. fatal(): Write a message and terminate ............................................................ 176
31. Epilogue ................................................................................................................ 176
32. Module uidfinder.py .......................................................................................... 176
   32.1. uidfinder: Prologue ......................................................................................... 177
   32.2. uidfinder.py: Imports ..................................................................................... 178
   32.3. uidfinder.py: Manifest constants .................................................................... 178
   32.4. uidfinder.py: Specification functions .............................................................. 182
   32.5. class UidFinder: The name lookup widget ....................................................... 182
   32.6. UidFinder.__init__() .......................................................................................... 183
   32.7. UidFinder._createWidgets(): Create all widgets and control variables ....... 184
   32.8. UidFinder._populate(): Find all TCC accounts that can have Moodle accounts ... 185
   32.9. UidFinder._ldapQuery() ................................................................................. 186
   32.10. UidFinder._clearHandler() ............................................................................. 187
   32.11. UidFinder._findByUid(): Look up a specific user ID .................................... 188
   32.12. UidFinder._clearList(): Clear out the list ................................................... 188
   32.13. UidFinder._listAdd(): Add one entry to the list ........................................... 188
   32.14. UidFinder._nameSearch(): Generic name search ....................................... 189
   32.15. uidfinder.py: UidFinder._selectHandler(): Handle a click on a line in the listbox .................. 190
   32.16. uidfinder.py: UidFinder.get() ....................................................................... 191
   32.17. uidfinder.py: UidFinder.lookupUid() ............................................................. 192
   32.18. uidfinder.py: UidFinder.clear() ..................................................................... 192
   32.19. uidfinder.py: class TccAccount ................................................................. 192
   32.20. uidfinder.py: TccAccount.__init__() ............................................................. 193
   32.21. uidfinder.py: TccAccount.__cmp__() ............................................................. 193
   32.22. uidfinder.py: TccAccount.__str__() ............................................................. 194
   32.23. uidfinder.py: Test driver ............................................................................... 194
   32.24. uidfinder.py: Epilogue ................................................................................... 195
33. Testing .............................................................................................................. 196
   33.1. Tests for user story A ....................................................................................... 196
   33.2. Tests for user story B ....................................................................................... 197
   33.3. Tests for user story C ....................................................................................... 198
1. Workflow for CMS course and section creation

The purpose of the cmsadds graphical user interface (GUI) is to streamline the workflow of the administrator of New Mexico Tech’s Course Management System (CMS) in creating courses.

You should be familiar with these related documents:

• *NMT Moodle 2.2 Administrator’s Guide*[^3]: This is the basic procedures manual for administering our current Course Management System, currently Moodle 2.2.

• *cmsimport2: Courseware Banner integration tools*[^4]: Describes the suite of programs used by Moodle administrators, and their implementation.

Here is an overall outline of the course creation process.

1. When an instructor wishes to establish one or more Moodle sites for their courses, they send mail to <cms_master@nmt.edu>. These emails do not always contain all the information you will need, so in many cases, your job as an administrator is to continue to correspond with the instructor to clarify exactly what they want.

2. Once you have enough information, run cmsadds. At this point it runs only under Linux, but it would not be much work to supply a Windows version if you should prefer that platform.

The cmsadds interface will help you find and confirm the identities of the instructor(s) and the exact Course Reference Numbers (CRNs) and other information on the current set of scheduled class sections.

You will maintain a separate file for each semester. The general form of the name of this file is:

```plaintext
adds-season-yyy.xml
```

*season*

Either summer, fall, or spring.

*yyy*

The four-digit calendar year.

**Warning**

The Registrar defines an “academic year” that begins in summer and ends in spring. In the author’s opinion, it is counterintuitive (in the summer and fall terms, the academic year is one greater than the calendar year), and thus prefers to use the calendar year, which everyone always knows.

3. When you terminate the cmsadds application, it will update the corresponding *adds-season-yyy*.xml file. To transload the updated course definitions into the CMS, refer to *NMT Moodle 2.2 Administrator’s Guide*.[^5]

2. Glossary

2.1. Section

The basic unit of coursework at New Mexico Tech is the section. Each section is uniquely identified within a semester by its five-digit Course Reference Number (CRN). (CRNs may be recycled in other semesters.)

Every section also has a unique identifier of this form:

\[
\text{dept} \quad \text{CCC}[A|B][T]-SS
\]

- **dept**
  - The two-to-four-character department code, e.g., “CH E”.

- **CCC**
  - A three-digit course number.

- **[A|B]**
  - Certain course numbers may have a suffix letter A or B.

- **T**
  - An optional section type code, such as “L” for lab. Omitted for lecture sections.

- **SS**
  - A two-digit section number.

Examples of section identifiers:

<table>
<thead>
<tr>
<th>Course</th>
<th>CRN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHEM</td>
<td>121-03</td>
</tr>
<tr>
<td>PHYS</td>
<td>189L-01</td>
</tr>
<tr>
<td>CH E</td>
<td>408-01</td>
</tr>
<tr>
<td>ST</td>
<td>550AD-01</td>
</tr>
</tbody>
</table>

2.2. Course

The term “course” is ambiguous.

- For the New Mexico Tech Registrar, a course is defined as specific department code and three-digit course number, e.g., “CHEM 121”.

- **Moodle course = NMT section.** Moodle’s basic organizational unit is the course. Unlike the previous Blackboard CMS, Moodle does not have an entity corresponding to a New Mexico Tech course.

  Most Moodle courses will correspond to exactly one NMT section. However, because of cross-listed courses, the Moodle administrator may also create Moodle courses to represent a cross-listed set; see Section 2.3, “Metacourse” (p. 8).

2.3. Metacourse

When multiple sections are cross-listed in the class schedule (for example, IT 121 and CSE 121 meet together), the corresponding Moodle construct is the metacourse. A metacourse inherits all the student enrollments from its child sections, and gives the instructor a place to post content shared between the nominal sections.
2.4. Cross-listed

The term *cross-listed* refers to exactly the same scheduling situation as described above in Section 2.3, “Metacourse” (p. 8).

In this implementation, there are a number of names related to this feature that may use either term.

- In the `adds.rnc` schema, the XML element that describes the request for one metacourse is `xlist`. In the `readadds` module that is used to read files with this schema, the corresponding class is `CrossParent`.
- In most other places, names of related entities generally use the word “meta”. Examples: in this application, classes `MetaList` and `MetaPanel`.

The designer (J. Shipman) apologizes for this confusion. The applications that work with Moodle are updated, rewritten versions of applications that used to work with the now-discarded Blackboard and WebCT courseware systems, and those systems consistently referred to the entity as “cross-listed sets”. Only with Moodle did the term “metacourse” creep in.

3. Design notes

This application is an example of lightweight literate programming: this document contains the actual code, which is mechanically extracted for execution. It was developed using the Cleanroom software development protocol.

3.1. Required output

Referring to the `adds.rnc` schema from the CMS import suite, we find that we need these items of information to create a Moodle course for an NMT section:

- The host name is currently always “Moodle”.
- The academic year and semester code. If we make the `adds-season-yyyy.xml` file name a command line argument to the `cmsadds` script, we can infer these items from the name of the file.
- The instructor’s TCC account name (UID). This is necessary because the `adds.rnc` schema organizes section requests by this key, so that all the sections requested by one instructor appear together on one area in the GUI.

If the email requesting the course came from there, or if you know their email, the application will provide an entry field so you can enter it there and jump directly to that person’s GUI area.

If the email came from off-campus, the application will provide a facility that allows you to search for their names in the set of accounts defined by the TCC LDAP server, and look up the corresponding TCC user ID.

- The section identifier containing CRN, department code, course number and title, section type, section number, and section title.

If you know the CRN, `cmsadds` will provide an entry field where you can enter it directly, and the application will fill out all this information from the schedule data made available by the `cron` job described in *TCC Public Data Project: Class schedules*.

---

6 http://www.nmt.edu/~shipman/soft/litprog/
7 http://www.nmt.edu/~shipman/soft/clean/
8 http://www.nmt.edu/tcc/doc/ihs/cmsimport2/
9 http://www.nmt.edu/tcc/data/class_sched/
If you don’t know the CRN, cmsadds will provide a page that displays a list of all the academic departments. When you select a department, the application will display a scrollable list of all the sections scheduled for the semester. Assuming that you have selected an instructor, you can add that section to that instructor’s page by clicking on it. The application will also provide a way to delete a section from an instructor’s page.

- Once the section has been added to an instructor’s page, you may then specify additional users to be added to that section in various roles. Refer to the adds . rnc schema definition for the roles that can be added.

Again, you will need to know the TCC UID to add a role to a section, so the features for searching by name or UID must be available for these functions as well.

There is one role for which you won’t need to know the UID: the Advising Center role. The UID for this is constant, and defined in the administrator’s guide.

- There will be a place for you to enter information about reloading the course from a previous semester.

- The area for an instructor will be divided into three parts: general information like the instructor’s name at the top; a list of sections requested below that; and at the bottom of the page, a list of metacourses.

To create a metacourse, you will first turn on checkboxes for each section, then click a button. A popup menu will appear where you can specify the overall name for the metacourse, and optionally any information about reloading content from previous semesters. When you complete this popup, the metacourse will appear in the list of metacourses for that instructor.

The section entries in the metacourse table will have controls that allow you to delete that section. There will also be a control to delete an entire metacourse.

At this writing, metacourse creation is still manual. These features are provided for your record-keeping, so you can keep track of what metacourses you have created manually.

The application will have a button that updates the adds - season - yyyy . xml file and exits. It is recommended that you use some source code control utility such as git or RCS to archive successive increments, to provide backout and disaster recovery.

3.2. User stories

As a preliminary to the actual widget-level design, here are some user stories, told from the point of view of the CMS administrator. These stories help us visualize what information and controls must be visible in the application as the workflow progresses. These examples were taken from actual email requests, slightly altered, using a fictional department code.

3.2.1. Story A: Single section, single instructor

Could you create a Moodle section for CERA 519? (Sent from a TCC account)

Perusal of the schedule shows only one section, lecture CERA 519-01, and the instructor name matches the requestor. Everything we need is in the schedule.

http://www.nmt.edu/tcc/help/pubs/moodle22/mooadmin22/
3.2.2. Story B: Metacourse

Please add these courses: CRN 23232, CERA 123-01, Engineering Headgear. The following two sections should be merged as one course: CRN 23456, CERA 489-01, Bridge Cozies; and CRN 23457, CERA 589-01, Bridge Cozies.

The mail came from a TCC account, so again we have everything we need from the class schedule.

3.2.3. Story C: What sections, who is this new instructor?

I need moodle for four courses, two of which should be combined for one Moodle course.

CERA 343, Kettle Blackening, CRN 43210.
CERA 343L, Kettle Blackening Lab, CRN 43211.
CERA 437 and CERA 537, Glazes and Phases, CRNs 43234 and 43235. I’m team-teaching with new professor Jane Janes, jjjjj@gmail.com, who should have full access.

The first problem is that it’s not entirely clear which pairs of sections should be combined. It’s likely that the requestor meant the second pair, since they are combined in one paragraph, but a followup email would not be out of line.

The second problem is that you need a TCC email for the new professor. Because you have no control over when she gets her new account, we suggest you send her mail at her gmail account and ask her to send you the TCC email when she gets it.

3.2.4. Story D: Who are these TAs?

Please set up a Moodle page for CERA 203, CRN 46820. I am one instructor; the other, Bill Billy, has no contract yet. TAs will be Dave Davies and Earl Early. Can we please transfer all files from last spring? (From a non-TCC account)

The first problem is to find the TCC account of the requestor. The class schedule should show the same name as the return address in the requesting email; if not, send a followup to clarify who is the primary instructor and what other instructors there are, if any. Once you are clear on the instructor’s name, cmsadds will provide a way to search for a given string through the names in the TCC LDAP directory.

Finding TCC accounts for the TAs will use that same search facility. If the searches fail, perhaps because the TAs are new and don’t have TCC accounts yet, send a followup to the requestor, asking them to let you know when they have TCC accounts.

The last sentence of the request contains a vague reference to an earlier Moodle course whose content is to be copied. To find this course, you’ll look in the appropriate semester’s category within Moodle, and record the extended CRN (e.g., 54321.2012) of the old section to be copied.

3.2.5. Story E: Which sections do they mean?

For this spring, could you copy my CERA 131 and 231 from the fall and my 132 from the spring?

You go to the schedule and find that the department has multiple sections and multiple instructors for all three of these course numbers. Match the requestor’s name against the instructor names in the class schedule to find the CRNs and section numbers. Then derive the course content reload information as in the previous story.
3.3. Inputs

One primary input to this system will be the XML class schedule extracted from Banner by the system described in *TCC Public Data Project: Class schedules* .

For every instructor and other participant (TA, grader), we must know these things about them:

- Their TCC UID (email).
- The TCC account type. This is necessary because many people have multiple TCC UIDs with the same Banner ID. Hence, the name search results will display the TCC account type so that the CMS administrator can select the correct one.
- Their full name.

All the above attributes are available with an anonymous binding to the TCC LDAP.

3.4. General workflow

Here is a state diagram for operations with *cmsadds*.

**semester page**
Shows the currently selected semester and allows the user to select a different semester. The set of semester schedules will determine which semesters they may select. For the default semester that will be displayed, see Section 5, “Operation” (p. 19).

**instructor page**
Displays all the sections and metacourses for one instructor.

**department page**
If the instructor has not provided a CRN, or if you want to check the CRN, select a department, which brings up all the scheduled sections in that department for the selected semester.

**preview CRN**
Shows all the information about one section so you can see if it is the one you want to add to this instructor.

---

11 http://www.nmt.edu/tcc/data/class_sched/
section edit
Shows all the information about one of the instructor’s sections, and allows editing of all the related information except for role assignments.

gerades page
Shows all the role assignments for one section, other than the primary instructor (who always has the Teacher role).

metacourse page
Shows all the sections of one metacourse.

The next step in design is to determine what information has to be visible in the GUI for each state in the above diagram.

3.5. Design of the UID finder

Teachers and other roles are connected to Moodle via their TCC account name (UID, User ID). However, when the requestor of a course does not supply a UID, we may have only the name that appears in the “From:” field of their email.

Even when the requestor provides UIDs for TAs and co-teachers, we would like to be sure that the UIDs given match their names.

So the UI must provide a good way to look up UIDs based on names. One discarded possibility is to provide a search on surnames. That won’t work: if the name has three or more words in it, the CMS administrator may not necessarily know where the surname starts. Consider surnames such as “ter Horst” or “de la O” or “Garcia y Vega”.

Another problem is that one person may have multiple UIDs linked to the same Banner ID (for example, a User Consultant who has both an Undergraduate account and a Staff account).

So the UI will allow the administrator to search on any whole word in the name. The names will come from the TCC LDAP account structure. Entries here have these relevant fields:

uid
The UID, also known as the account name.

cn
A string consisting of the UID, followed by one space, followed by the person’s name as “First [Middle] Last”. (There is another, optional attribute called “gecos” that is supposed to be the user’s real name, but users may change it to something silly.)

tccAccountType
There are numerous account types, but the ones that are valid for use in instruction are emeritus, faculty, graduate, special, undergrad, and undergraduate.

In the GUI, this search facility will be represented as a Tkinter LabelFrame widget with the label “UID finder”. Within this frame will be these widgets:

• A vertically scrollable Listbox.

• An Entry field labeled “UID” and an accompanying button labeled “Search”. When that button is clicked, or the user presses Enter, if the UID in the field is defined in the TCC LDAP, the listbox will be cleared except for the first line, which will display the account type, UID, and full name in that order. That line of the listbox will also be the selected line.

The account types will be abbreviated according to this table:
<table>
<thead>
<tr>
<th>Code</th>
<th>Account type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Distance Ed</td>
</tr>
<tr>
<td>Em</td>
<td>Emeritus</td>
</tr>
<tr>
<td>Fa</td>
<td>Faculty</td>
</tr>
<tr>
<td>Gr</td>
<td>Graduate</td>
</tr>
<tr>
<td>MS</td>
<td>MST</td>
</tr>
<tr>
<td>Sp</td>
<td>Special</td>
</tr>
<tr>
<td>St</td>
<td>Staff (Payroll)</td>
</tr>
<tr>
<td>Sn</td>
<td>Staff (non-payroll)</td>
</tr>
<tr>
<td>UG</td>
<td>Undergrad</td>
</tr>
</tbody>
</table>

- An Entry field labeled “Name search” and an accompanying button labeled “Search”. When that button is clicked, or the user presses Enter, if the UID in the field is defined in the TCC LDAP, the listbox will be cleared and its contents replaced with a list of all the names that match. These lines will have the same format as the UID search: account type, UID, and full name in that order.

If $S$ is the contents of the search field, then a person's name is considered to match if any word in that name starts with $S$. For example, if the search string is “ter”, that will match “Rita ter Horst” and also “Terence Smith”. The matching will be case-insensitive.

In the listbox, if the first name matches, the displayed name will be the full name. However, if one of the other words in the name matches, the displayed name will be permuted so that the matching word is at the front, followed by a comma, followed by the words preceding the match. Here are some examples for a match against “Te”, following the account type codes and UID:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UG</td>
<td>ctenboom Ten Boom, Clara</td>
</tr>
<tr>
<td>Gr</td>
<td>rterhors ter Horst, Rita</td>
</tr>
<tr>
<td>Fa</td>
<td>tsmith00 Terence Smith</td>
</tr>
<tr>
<td>Sp</td>
<td>blastor Terry James, Joe</td>
</tr>
</tbody>
</table>

The connections between this widget group and the rest of the application may take two forms:

- The constructor will have an optional callback argument, a procedure which will be called when the user clicks on a line in the listbox.

- Other widgets such as buttons may query the state of the listbox. If a line is selected, the query will return an instance containing the account type, UID, and name. If no line is selected, the query will signify this as an error condition.

3.6. The archive subdirectories: Accountability considerations for uploading

The purpose of the cmsadds application is to maintain a set of working files, one for each semester, each named as “adds-season-yyyy.xml”. However, nothing happens on the CMS side until the file is run through the import tools and the resulting file uploaded into the CMS.

Because we have created our own integration software here, the author is concerned that if a problem occurs, we know exactly what file was uploaded. Hence, every time a working file is saved, a copy will be made in a subdirectory of the current working directory, containing a timestamp as part of its name. This will also allow backtracking in the case of disaster.
There is one archive subdirectory for each semester, and its name has the form `arch-season-yyyy`. The archive copy will have a timestamp string of the form `yyyy-mm-ddThh-mm-ss` appended to its name.

Also, when the user has made any changes, the application will not allow them to terminate until the changes are saved to the working file, or unless they specify that all the changes are to be discarded.

4. General layout

Here is the overall layout of the application on the display.

In the “general controls” area are buttons that must always be available such as *Save* and *Quit*. A prominent feature of this area is an instance of the *UidFinder* widget discussed in Section 3.5, “Design of the UID finder” (p. 13) bearing the title *Select primary instructor*: when the user selects a user ID here, the area labeled “primary instructor panel” displays all the data for that instructor.

The area labeled “swap area” may display different transient panels at different times.

- Initially, this area is blank. It may become blank again.

- When the user is searching for a section to add for this instructor, they can select an academic department and the swap area will display a *department schedule panel* showing the sections scheduled for the currently displayed semester.

- In the primary instructor panel there is a list of that instructor’s sections that are currently in the CMS. If the user clicks on one of these sections, a *section edit panel* appears in the swap area. Here the user can add or delete roles, or edit the section name.

- Also in the primary instructor panel is a list of that instructor’s metacourses. If the user clicks on a metacourse in that list, or clicks the *Add metacourse* button, a *metacourse edit panel* appears in the swap area. Here, there are controls to add or delete the metacourse’s child sections.

In the following sections we will enumerate all the widgets and their classes according to their location in these panels: the two fixed panels and the ones that can appear in the swap area.

- Section 4.1, “Layout of the general controls panel” (p. 16).
- Section 4.2, “Layout of the primary instructor panel” (p. 17).
- Section 4.3, “Layout of the department schedule panel” (p. 18).
- Section 4.4, “Layout of the section edit panel” (p. 18).
- Section 4.5, “Layout of the metacourse panel” (p. 19).
4.1. Layout of the general controls panel

These controls will always be visible.

Page title, ttk.Label
Identifies the application.

Click to show an instructor's courses, UidFinder
For the design of this compound widget, see Section 3.5, “Design of the UID finder” (p. 13). When the user selects a user ID using this widget, the primary instructor panel displays the existing course information for that user ID, if any. If the input file does not have a req element for that email, empty lists of sections and metacourses will appear.

Change semester, ttk.LabelFrame.
Contains a ttk.MenuButton that displays the names of available semesters. Use this drop-down menu to change to a different semester.

Save adds-season-yyyy.xml, ttk.Button
See Section 5, “Operation” (p. 19) for the semantics of the Save button.

Upload cms accounts
Runs cmsaccts from the cmsimport suite, saves the file in the appropriate archive subdirectory, and uploads the file into the CMS.

Because the accounts list comes directly from the TCC LDAP directory tree, it doesn’t matter whether or not you have saved your changes to the adds-season-yyyy.xml file.

Upload cms courses, ttk.Button
If the current tree is dirty (see Section 5, “Operation” (p. 19)), the user will get a popup with choices Save adds-season-yyyy.xml or Cancel.

If the tree is clean or the user selects Save, first a copy of the current adds-season-yyyy.xml file is written to an archive subdirectory, so we will have a record of what was uploaded; see Section 3.6, “The archive subdirectories: Accountability considerations for uploading” (p. 14).

Then cmsadds will run the cmssects application to create a sects_... IMS import file, make a copy of that file in the archive subdirectory, and then import that file into the CMS.

Upload cms enrollments
If the current tree is dirty (see Section 5, “Operation” (p. 19)), the user will get a popup with choices Save adds-season-yyyy.xml or Cancel.

If the tree is clean or the user selects Save, first a copy of the current adds-season-yyyy.xml file is written to an archive subdirectory, so we will have a record of what was uploaded; see Section 3.6, “The archive subdirectories: Accountability considerations for uploading” (p. 14).

Then cmsadds will run the cmsenroll application to create an enroll_... IMS import file, make a copy of that file in the archive subdirectory, and then import that file into the CMS.

Cleanup cms drops
Runs the mooroster application with the --bancheck and --write options to generate an IMS import file that will unenroll students that are currently in Moodle (for the semester currently selected in cmsadds), but who are no longer enrolled in that section according to Banner.

Then it will make a copy of the resulting cleanup_... file in the appropriate archive subdirectory, and upload that file to Moodle.

Because the cleanup file is the result of a comparison between Banner enrollments and the underlying Moodle database, it doesn’t matter whether or not you have saved your changes to the adds-season-yyyy.xml file.
Quit, ttk.Button
See Section 5, “Operation” (p. 19) for the semantics of this button: we don’t want to make it easy for the user to make changes and then discard them.

4.2. Layout of the primary instructor panel

Here are the widgets used to manage all the course requests for a specific primary instructor.

Primary instructor, ttk.Label
Displays that text plus the user ID and full name of the currently selected instructor.

Edit instructor name, ttk.LabelFrame
Groups the next six controls.

Given, ttk.Entry and ttk.Label
Allows the editing of the given (first) name.

Middle, ttk.Entry and ttk.Label
Allows the editing of the middle name.

Family, ttk.Entry and ttk.Label
Allows the editing of the family name (surname).

Remove this instructor from the CMS, ttk.Button
If the instructor has any sections in the CMS, the user will get a popup informing them of that. One button will read Discard all sections for (instructor)? If they click this, the instructor’s req element will be removed from the tree. The other button will read Cancel.

If an instructor has no sections displayed when the cmsadds application terminates, they will not have a corresponding req element in the tree, even if they had one previously. In this case it is not necessary to click this Remove button.

Close, ttk.Button
Closes any associated page in the swap area and blanks out the primary instructor panel.

Sections: right-click to edit, ttk.LabelFrame
A wrapper for all controls related to maintaining the instructor’s list of sections.

Section list, tk.Listbox and vertical ttk.Scrollbar.
Each line displays three items: the CRN; the composite section number, e.g., “CH E 589L-01”; and the section title. If the user right-clicks on a nonempty line of this listbox, it the section edit panel for that section appears in the swap area. We use right-click for this function, because left-click is used to select a section for addition to a metacourse.

Find CRN, ttk.Entry and associated ttk.Button
If the user clicks the button or presses Enter in the entry, and the contents of the entry are a CRN defined in the current semester’s schedule, the line for that section is added to the section list. Otherwise the user gets a No such CRN popup.

Delete selected section, ttk.Button
To remove a section from the instructor’s set, the user clicks on the section line in the listbox and then clicks the button. An Are you sure? popup will require confirmation.

Select department, ttk.LabelFrame
Contains various ways to select a department and display its schedule.

Find department, ttk.Button, and accompanying ttk.Entry
To select a department by its four-letter code, enter that code in the Entry and then either press Enter or click the button.
**ttk.Menu**
A dropdown menu that shows all the department codes and names. The user may use this to select a department.

**Metacourses: right-click to edit, ttk.LabelFrame**
A wrapper for all controls related to maintaining the instructor's set of metacourses.

**Add metacourse, ttk.Button**
Displays the metacourse panel in the swap area, with an empty name and no sections added.

**Delete selected metacourse, ttk.Button**
To delete a metacourse, select its line in the listbox and then click this button. An *Are you sure?* popup will ask for confirmation if the metacourse list still has any sections in it.

**Metacourse list, tk.Listbox**
One line for each of the instructor's metacourses, displaying the title. If the user right-clicks on a nonempty line in this box, its metacourse panel will appear in the swap area. We use right-click for this function because left-click is used to select a metacourse for deletion.

### 4.3. Layout of the department schedule panel

Here are the widgets in the department schedule panel.

**Panel title, ttk.Label**
Displays the department code, department title, and current semester and year.

**Add selected section to this instructor, ttk.Button**
If an entry in the section list is selected, the corresponding section is added to the section listbox in the primary instructor panel. If that CRN is already in the list, the user will get an error popup.

**Close, ttk.Button**
Removes this panel from the swap area, leaving it blank.

**Section list, tk.Listbox with associated horizontal and vertical ttk.Scrollbars**
The box contains one line for each section in this department's schedule for the currently selected semester. Each line contains: the CRN; the full section number, e.g., "MUS 321-01"; the section title from the official schedule; and the names of any instructors in parentheses.

### 4.4. Layout of the section edit panel

One purpose of the section edit panel is to allow editing of the section title, because the versions from the official schedule are often heavily curated, e.g., “Inst., Meas., & Proc. Cont. Lab I”. Another purpose is to manage role assignments (other than the primary instructor's automatic role assignment).

**Page title, ttk.Label**
Says “Instructor:” followed by the primary instructor's user ID and full name. On a separate line is the section's CRN and full section number.

**Title, ttk.Entry and associated ttk.Label**
When a section is first brought in from the official schedule, its title appears here. We encourage the user to expand it to something more readable by editing it in this entry.

**Close, ttk.Button**
If the data in the panel are in a valid state, removes the section edit panel and blanks the swap area. Example of an invalid state: the section title is empty. In that case, a popup will appear with these choices: *Return to editing section name* or *Discard this entire section*. 
Reload from, ttk.Entry and associated ttk.Label
A place to record the instructor’s request to reload content from an earlier semester.

Roles, ttk.LabelFrame
A wrapper for the display of currently selected roles.

Role table, tk.Listbox and associated vertical ttk.Scrollbar
For each role, displays the full role name (“Instructor”, “Non-editing teacher”, or “Grader”), and the roleholder’s user ID and full name.

Delete selected role, ttk.Button
If a line of the listbox is selected, this button removes that person from that role, after confirming via an Are you sure? popup.

Advising Center access, ttk.Checkbutton
If checked, the “Advising Center” role is given to the Advising Center’s user ID.

Add roleholder, ttk.LabelFrame
A wrapper for three buttons: Instructor, Non-editing teacher, and Grader. The user selects a user ID using the Select roleholder widget, then clicks the button for the desired role, which adds a line to the listbox of roles.

Select roleholder, UidFinder
See Section 3.5, “Design of the UID finder” (p. 13).

4.5. Layout of the metacourse panel
Widgets in the metacourse edit panel include:

Page title, ttk.Label
Displays the title Metacourse edit panel.

Title, ttk.Entry and associated ttk.Label
If the user is editing an existing metacourse, its title will appear here. If the panel appeared in response to the Add metacourse button, it will be empty.

If the user changes the contents of this entry, when focus leaves the widget, the corresponding line in the metacourse list in the instructor panel will be updated.

Child section list, tk.Listbox and corresponding vertical ttk.Scrollbar.
In this box appear lines for the child sections of this metacourse. Each line contains: the CRN; the full section number; and the section title.

Add selected section, ttk.Button
To add a child section to a metacourse, it must first appear in the primary instructor’s section list. This button adds the currently selected section from that list to the child section list.

Delete selected metacourse, ttk.Button
Removes the selected line in the child section list.

Close, ttk.Button
If the panel is in a valid state (a nonempty title and at least two child sections), blanks the swap area. If there is a problem, the user gets a popup with these choices: Return to the metacourse panel or Discard this entire metacourse.

5. Operation
To start up cmsadds at a Unix command line:
The first argument can be abbreviated to any unambiguous initial prefix; for example, “sp” is sufficient to specify the spring semester.

If the user does not specify the semester argument, the selected semester will default to these values depending on the current date.

<table>
<thead>
<tr>
<th>Date Range</th>
<th>Semester Starting Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec. 1–Apr. 30</td>
<td>Spring semester starting that January.</td>
</tr>
<tr>
<td>May 1–Jul. 30</td>
<td>Summer semester starting that June.</td>
</tr>
<tr>
<td>Aug. 1–Nov. 30</td>
<td>Fall semester starting that August.</td>
</tr>
</tbody>
</table>

If the user specifies a semester but not a year, the calendar year is derived this way:

- For summer semester, the calendar year is the current year up to September 1, then the following year.
- For fall semester, the calendar year is always the current year.
- For spring semester, the calendar year is the current year up to June 1, then the following year.

Here are some examples that show how the defaulting works.

<table>
<thead>
<tr>
<th>Command line</th>
<th>Today’s date</th>
<th>Selected semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmsadds</td>
<td>2013-12-01</td>
<td>Spring 2014</td>
</tr>
<tr>
<td>cmsadds</td>
<td>2014-04-30</td>
<td>Spring 2014</td>
</tr>
<tr>
<td>cmsadds</td>
<td>2014-05-01</td>
<td>Summer 2014</td>
</tr>
<tr>
<td>cmsadds su</td>
<td>2013-08-31</td>
<td>Summer 2013</td>
</tr>
<tr>
<td>cmsadds su</td>
<td>2013-09-01</td>
<td>Summer 2014</td>
</tr>
<tr>
<td>cmsadds f</td>
<td>2014-05-01</td>
<td>Fall 2014</td>
</tr>
<tr>
<td>cmsadds sp 2018</td>
<td>any</td>
<td>Spring 2018</td>
</tr>
</tbody>
</table>

When cmsadds starts, it will look for a working file whose name has the form adds-season-yyyy.xml, according to the selected semester. If there is no such file, the user will get a popup that gives them these choices:

- **Create new file**: When the user is just starting to build a course set for the new semester, this proceeds into the application with a completely empty set of courses.
- **Cancel**: Quits.

This file will be converted into an internal tree. At this point the application will make a complete saved copy of that tree, as it represents the state of things last time the working file is saved. Whenever the user clicks the **Save** button, a copy of the current tree will again be saved.

Whenever the user either clicks the **Quit** button or changes the current semester, the current tree will be compared to the saved copy to insure that the last-saved adds-season-yyyy.xml file has all the changes. If the trees differ, the user will get a popup asking them to either **Save adds-season-yyyy.xml** or **Discard all changes**.

We refer to the current tree as **clean** if it is identical to the last-saved tree, or **dirty** if the user has made any changes since the last save.
6. Object architecture

In order that our structure have some logic to it, avoiding the syndrome where every Tkinter widget may potentially need to interact with every other widget, we group our widgets into compound widgets, each represented by a class. There is a hierarchy from large aggregations of widgets at the higher levels, subdivided into smaller groupings until we reach the level of individual widgets.

Here, in outline form, is the set of classes that make up the Tkinter application.

App
The application as a whole. This instance manages the `et.ElementTree` that represents the external XML file.

UidFinder
A compound widget that lets a user select a TCC account either by its user ID (account name) or by name searching. See Section 32, “Module uidfinder.py” (p. 176).

ReqPanel
Primary instructor panel: this panel manages the presentation and editing of the `req` XML element that contains all the section requests for one primary instructor.

SectionsList
The compound widget displaying the current list of sections with associated controls.

MetaList
The compound widget displaying the current list of metacourses with associated controls.

SwapArea
The swap area, a `ttk.Frame` used to present different panels at different times.

SchedPanel
The department schedule panel; displays in the swap area.

MetaPanel
The metacourse panel; displays in the swap area; corresponds to the `xlist` XML element that defines a metacourse.

SectionPanel
The section edit panel; displays in the swap area; corresponds to a `section` XML element.

RolesList
A compound widget containing the list of current roleholders and associated controls.

UidFinder
Another copy of the widget described in Section 32, “Module uidfinder.py” (p. 176), for selecting roleholders.

Another way of looking at this design: there is a `model-controller` relationship between instances of the classes from the `readadds.py` module and our groupings of widgets:

<table>
<thead>
<tr>
<th>Model</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>readadds.AddsFile</td>
<td>App</td>
</tr>
<tr>
<td>readadds.Teacher</td>
<td>ReqPanel</td>
</tr>
<tr>
<td>readadds.Section</td>
<td>SectionPanel</td>
</tr>
<tr>
<td>readadds.CrossParent</td>
<td>MetaPanel</td>
</tr>
</tbody>
</table>
6.1. Control flow

Among the subclasses inside the overall **App** class, our design follows the Law of Demeter[^12] or Principle of Least Knowledge. Specifically, a widget should use method calls on its immediate neighbors, never making any assumption about the neighbors of its neighbors.

For example, when a user clicks on a section in the list of sections in the primary instructor panel (**ReqPanel**), the application must display the corresponding section edit panel. In terms of actual method calls:

1. Logic in **ReqPanel** calls a method in the **App** instance, asking it to display a **SectionPanel**.
2. The **App** instance calls a method in the **SwapArea** instance asking it to display a **SectionPanel**.
3. The **App** instance calls a method in the **SectionPanel** to display the XML section node.

The above sequence is slightly oversimplified. One complication is that there may be a panel already present in the swap area that is in an invalid state, such as a metacourse with no child sections. Here is a scenario showing what might happen. Suppose the primary instructor panel is currently displaying a **req** element and the swap area is blank.

1. In the primary instructor panel, the user clicks the **Add metacourse** button. The metacourse panel appears in the swap area with a blank title and no child courses. The user adds one child course and then gets distracted.
2. Back in the primary instructor panel, the user clicks on a section in the section list. This is a request to display the section edit panel in the swap area, but the swap area currently shows a half-finished metacourse.

Each of the transient panels that can appear in the swap area will have a `.close()` method that requests it to remove itself from the swap area. When the `.close()` method is called in this case, it notices that the metacourse is not in a valid case and presents the popup described in Section 4.5, “Layout of the metacourse panel” (p. 19), with choices *Return to the metacourse panel* or *Discard this entire metacourse*.

If the user chooses *Discard...*, the `.close()` method removes the associated **xlist** element from the XML tree and returns.

However, if the user chooses *Return...*, the `.close()` method raises a **CloseFailure** exception. Any application logic that needs to use the swap area must anticipate this exception and, if it occurs, abandon whatever operation it is trying to perform.

Hence, in general each transient panel will have a `.close()` method that can fail in a similar way.

Let’s look now at the interactions between the top-level classes during various control actions.

6.2. Requests originating from **App**

Here are the control flows for controls in the **App** class (the general controls panel).

- **Select primary instructor.** **ReqPanel**: Display this **req** XML element.
- **Change semester.** Calls the `.close()` method for the panel currently in the swap area, if any, and for the **ReqPanel**, if it is currently displaying an instructor. It then checks to see if the current set of requests has changed since the last save; if not, the newly selected semester will appear in the semester selector widget.

If there are unsaved changes in the current request set, the user gets a popup with two choices: discard the changes or return to the current semester.

6.3. Requests originating from ReqPanel

These control flows start from controls in the primary instructor panel.

• Remove this instructor from the CMS. Asks App to remove this req element altogether.
• User clicked on a section in the sections list. Asks App to close the swap area and then display a SectionPanel.
• Show dept schedule. Asks App to close the swap area and then display a specific department schedule.
• Add metacourse. Asks App to close the swap area and then display an empty MetaPanel.
• User clicked on a metacourse in the metacourse list. Asks App to close the swap area and then display the selected xlist element.

6.4. Requests originating from SchedPanel

• Add selected section to this instructor. Passes schedule information to App, which asks the ReqPanel to add it to its section list.

6.5. Requests originating from MetaPanel

• Add selected section. Asks App to ask ReqPanel if a section is currently selected in its section list. If so, adds that section to the metacourse. If no section is selected, displays an error popup.

• A MetaPanel may request the parent App to delete its currently displayed metacourse in this situation:
  1. The MetaPanel.close() method is called, whether because the user clicked the Close button, or because some other widget needs to use the swap area.
  2. The displayed metacourse is not in a valid state, for example because it has no title. (See Section 26, “class MetaPanel: Edit cross-listed sets (metacourses)” (p. 162) for a full list of validity constraints.)
  3. The user gets a popup asking if they want to discard the half-built metacourse, and they click on Yes.

If all these conditions are met, the MetaPanel asks the App to delete from its model the currently displayed readadds.CrossParent instance, and blank the transient area. The App delegates the change to the model to the ReqPanel, because that panel also needs to update its list of metacourses (the MetaList widget group).

6.6. Requests originating from SectionPanel

At this writing, the SectionPanel class has no control actions that impact outside classes. However, it does interact with its contained RolesList widget.

When the user clicks one of the Buttons in the group Add selected roleholder as, the SectionPanel widget queries its contained UidFinder. If an account within the UidFinder is selected, the SectionPanel asks its RolesList to add that account with the role corresponding to the clicked button.
7. Code prologue

Here begins the actual code for the application. Python requires that imports from the __future__ module appear before any other executable code.

```python
#!/usr/bin/env python
#================================================================
# cmsadds: GUI editor for CMS course creation.
#
# Do not edit this file directly. It is mechanically extracted
# from the documentation:
#   http://www.nmt.edu/tcc/projects/cmsadds/
#---------------------------------------------------------------
from __future__ import print_function
```

8. Module imports

```python
# - - - - - I m p o r t s

The standard Python sys module gives us access to the command line options and standard I/O streams.

```python
import sys
```

The Args class uses the Singleton base class from Logging, scanning, and singleton objects for Python: logscan.py and singleton.py.

```python
import singleton
```

The Args class also uses the standard Python argparse module to handle initial parsing of command line arguments.

```python
import argparse
```

The standard Python re module provides regular expression matching.

```python
import re
```

Next we import the standard Python date and time module.

```python
import datetime
```

We need the standard Python os module for the os.path.exists() function.

```python
import os
```

We use the pexpect module to run cmssects in a subprocess and capture the output.

```python
import pexpect
```

The `readadds` module comes from *cmsimport2: Courseware Banner integration tools*.

```python
import readadds
```

Class schedule data comes from *TCC Public Data Project: Class schedules*.

```python
import class_sched
```

For XML operations, including reading, writing, and updating, see *Python XML processing with lxml*. Name `et` is the `etree` module; `E` is an element factory; and we also use the `subElement()` function to add child elements.

```python
from etbuilder import E, et, subElement
```

The Tk widget set comprises four modules: `Tkinter` is the basic widget set, including constants; `ttk` supplies “themed” widgets; `tkFont` supplies a full set of text fonts; and `tkMessageBox` supplies error popups.

```python
import Tkinter as tk
import ttk
import tkFont
import tkMessageBox
```

The `tkSimpleDialog` module is part of the standard Python install; see its documentation.

```python
import tkSimpleDialog
```

For the `uidfinder` module, see Section 32, “Module `uidfinder.py`” (p. 176).

```python
import uidfinder
```

## 9. Manifest constants

Since Python does not have constants *per se*, we use a naming convention: names of constants are in all capital letters with “_” used as a word separator.

```python
# - - - - - M a n i f e s t c o n s t a n t s
```

### 9.1. PROGRAM_NAME

Name of the program.

```python
PROGRAM_NAME = 'cmsadds'
```
9.2. **VERSION**

Current version identifier.

```python
VERSION = '1.0.3'
```

9.3. **SCHEMA_FILE**

The Relax NG schema that specifies the structure of the `adds-season-yyy.xml`.

```python
SCHEMA_FILE = 'adds.rnc'
```

9.4. **SCHED_FILE_PATH**

Full pathname of the schedule file that is input to the `class_sched.py` module.

```python
SCHED_FILE_PATH = '/u/www/docs/tcc/data/sched.xml'
```

9.5. **ADDS_PREFIX**

Prefix to the `adds-season-yyy.xml` file.

```python
ADDS_PREFIX = 'adds-
```

9.6. **XML_EXT**

Extension for the `adds-season-yyy.xml` file.

```python
XML_EXT = '.xml'
```

9.7. **ARCH_PREFIX**

Prefix for the subdirectory that holds archival copies of `adds-season-yyy.xml` files.

```python
ARCH_PREFIX = 'arch-
```

9.8. **TIME_TAIL_FORMAT**

A format string to be used by `datetime.datetime.strftime()` to format the current UTC as “yyy-dd-mmThh-mm-ssZ”.

```python
TIME_TAIL_FORMAT = "%Y-%m-%dT%H-%M-%SZ"
```

9.9. **ADVISING_EMAIL**

TCC user ID of the account that holds the Advising Center role within Moodle.
Note
Shipman records this value under protest. The account should be the organizational account for the Center for Student Success (css@nmt.edu), not that of an individual (Elaine DeBrine-Howell).

```python
ADVISING_EMAIL = 'elaine'
```

9.10. CMSACCTS_CMD
The shell command that runs cmsgacct.

```python
CMSACCTS_CMD = "cmsgacct"
```

9.11. CMSSECTS_CMD
The shell command that runs cmsgsect. It requires the adds-season-yyyy.xml file name as a .format() argument.

```python
CMSSECTS_CMD = "cmsgsect {0}"
```

9.12. CMSENROLL_CMD
The shell command that runs cmsgenroll. It requires the adds-season-yyyy.xml file name as a .format() argument.

```python
CMSENROLL_CMD = "cmsgenroll {0}"
```

9.13. MOOROSTER_CMD
The shell command that runs mooroster. The --bancheck argument requests that it check Banner against Moodle's internal database, and the --write argument requests that it write a cleanup... file that deletes dropped enrollments.

```python
MOOROSTER_CMD = "mooroster --bancheck --write"
```

9.14. OUT_FILE_LINE_PAT
A regular expression used to locate the line in the output of the various scripts that generate IMS files (such as cmsgsect) that specifies the name of the file that was written. This pattern depends on the Enterprise.write() method in the cmsgimport suite, and there is a note in that method documenting the dependency.

```python
OUT_FILE_LINE_PAT = re.compile(  
    r"Output file is " # Matches "Output file is "
    r"" # Start group 1
    r"[^']++" # Matches one or more non-single-quotes
)
9.15. **MOODLE UPLOAD_CMD**
The shell command that transfers a file to the Moodle IMS file import location.

```python
MOODLE UPLOAD_CMD = ('scp {0} 
   "cms@moodle.nmt.edu:/var/moodle/classes.xml"
```

9.16. **NEXT_SUMMER_DATE**
When summer semester is specified without a year, this is the (month, day) when we assume the summer of the following year.

```python
NEXT_SUMMER_DATE = (9, 1)
```

9.17. **NEXT_SPRING_DATE**
When spring semester is specified without a year, this is the (month, day) when we assume the spring of the following year.

```python
NEXT_SPRING_DATE = (6, 1)
```

9.18. **SPRING_SUMMER_DATE**
When no command line arguments are given, this is the (month, day) when we default to summer.

```python
SPRING_SUMMER_DATE = (5, 1)
```

9.19. **SUMMER_FALL_DATE**
When no command line arguments are given, this is the (month, day) when we default to fall.

```python
SUMMER_FALL_DATE = (8, 1)
```

9.20. **FALL_SPRING_DATE**
When no command line arguments are given, this is the (month, day) when we default to spring.

```python
FALL_SPRING_DATE = (12, 1)
```
9.21. **BASE_FONT_FAMILY**

Font family for most controls. Although both DejaVu Sans and Liberation Sans are available on all TCC Linux and Windows platforms, those fonts insufficiently distinguish between characters like I and 1 and l.

```python
BASE_FONT_FAMILY = 'Liberation Serif'
```

9.22. **NARROW_FONT_FAMILY**

```python
NARROW_FONT_FAMILY = 'Liberation Sans Narrow'
```

9.23. **MONOSPACE_FAMILY**

A monospaced font to be used in `ttk.Entry` widgets.

```python
MONOSPACE_FAMILY = 'DejaVu Sans Mono'
```

9.24. **ENTRY_STYLE**

The name of the `ttk` style for `Entry` widgets.

```python
ENTRY_STYLE = 'TEntry'
```

9.25. **FRAME_STYLE**

The name of the `ttk` style for `Frame` widgets.

```python
FRAME_STYLE = 'TFrame'
```

9.26. **LABEL_FRAME_STYLE**

The `ttk` style name for the `ttk.LabelFrame` widget.

```python
LABEL_FRAME STYLE = 'TLabelFrame'
```

9.27. **PANEL_TITLE_STYLE**

The `ttk` style name for the title label of a panel, with a large font.

```python
PANEL_TITLE_STYLE = 'Title.TLabel'
```

9.28. **MAIN_BG_COLOR**

Default frame background. Currently the standard color name OldLace.
9.29. MAIN_FG_COLOR
Default foreground color. Currently the standard TCC dark brown as in www.nmt.edu/tcc/style.css.

```
MAIN_FG_COLOR = '#663300'
```

9.30. ACTIVE_BG_COLOR
Default active background color (not the default gray): basically LimeGreen with a lot less saturation.

```
ACTIVE_BG_COLOR = '#88cc88'
```

9.31. REQ_WIDE
Width of the ReqPanel.

```
REQ_WIDE = 620
```

9.32. REQ_HIGH
Height of the ReqPanel.

```
REQ_HIGH = 700
```

9.33. SWAP_WIDE
Width of the SwapArea.

```
SWAP_WIDE = 520
```

9.34. SWAP_HIGH
Height of the SwapArea.

```
SWAP_HIGH = REQ_HIGH
```

9.35. GUI text constants
These constants define symbolic names for various text labels that appear in the graphical user interface.
9.35.1. **ADD_META_BUTTON**

Text of the button to add a new metacourse.

```
ADD_META_BUTTON = "Add metacourse"
```

9.35.2. **ADD_SECT_META_BUTTON**

Text on the button that adds the selected section to a metacourse.

```
ADD_SECT_META_BUTTON = "Add selected section"
```

9.35.3. **ADD_SEL_ROLE_FRAME**

Text for the frame around the group of buttons that add the selected roleholder in some role.

```
ADD_SEL_ROLE_FRAME = "Add roleholder"
```

9.35.4. **ADD_SEL_SECT_BUTTON**

Text on the button that adds a selected section to a metacourse.

```
ADD_SEL_SECT_BUTTON = "Add selected section to this instructor"
```

9.35.5. **ADVISING_CHECKBUTTON**

Text on the checkbox for Advising Center access.

```
ADVISING_CHECKBUTTON = "Advising Center access"
```

9.35.6. **CHANGE_SEMESTER_FRAME**

Text of the label frame around the drop-down menu to change semesters.

```
CHANGE_SEMESTER_FRAME = "Change semester"
```

9.35.7. **CLEANUP_DROPS_BUTTON**

Label on the button that disenrolls dropped students.

```
CLEANUP_DROPS_BUTTON = "Cleanup cms drops"
```

9.35.8. **CLOSE_BUTTON**

Text on the buttons that close various panels.

```
CLOSE_BUTTON = "Close"
```
9.35.9. **DEL_SECT_META_BUTTON**

Text on the button that deletes the selected section from a metacourse.

```plaintext
DEL_SECT_META_BUTTON = "Delete selected section"
```

9.35.10. **DEL_SEL_META_BUTTON**

Text on the button that deletes the selected metacourse.

```plaintext
DEL_SEL_META_BUTTON = "Delete selected metacourse"
```

9.35.11. **DEL_SEL_ROLE_BUTTON**

Text on the button that deletes a selected roleholder.

```plaintext
DEL_SEL_ROLE_BUTTON = "Delete selected role"
```

9.35.12. **DEL_SEL_SECTION**

Text on the button that deletes the selected section.

```plaintext
DEL_SEL_SECTION = "Delete selected section"
```

9.35.13. **EDIT_NAME_FRAME**

Name of the frame around the Entry widgets for editing the instructor's name.

```plaintext
EDIT_NAME_FRAME = "Edit instructor name"
```

9.35.14. **FAMILY_LABEL**

Label for the Entry for editing the instructor's surname.

```plaintext
FAMILY_LABEL = "Family"
```

9.35.15. **FIND_CRN_BUTTON**

Text on the button that searches for a Course Reference Number.

```plaintext
FIND_CRN_BUTTON = "Find CRN"
```

9.35.16. **FIND_DEPT_BUTTON**

Text on the button that searches for a department.

```plaintext
FIND_DEPT_BUTTON = 'Find department'
```
9.35.17. **GIVEN_LABEL**
Label for the Entry for editing the instructor's given name.

```python
GIVEN_LABEL = "Given"
```

9.35.18. **INSTR_LABEL**
Label on the primary instructor panel.

```python
INSTR_LABEL = "Primary instructor"
```

9.35.19. **MAIN_LABEL**: Main window label
Big title at the top of the main window.

```python
MAIN_LABEL = "cmsadds: CMS course request manager"
```

9.35.20. **META_DESC_LABEL**
Label for the entry widget for the description (title) of a metacourse.

```python
META_DESC_LABEL = "Title"
```

9.35.21. **META_FRAME**
Text of the frame around the set of metacourses.

```python
META_FRAME = "Metacourses: right-click to edit"
```

9.35.22. **META_PANEL_TITLE**
Title for the metacourse panel.

```python
META_PANEL_TITLE = "Metacourses"
```

9.35.23. **MIDDLE_LABEL**
Label for the Entry for editing the instructor’s middle name or initial.

```python
MIDDLE_LABEL = "Middle"
```

9.35.24. **QUIT_BUTTON**
Text on the button that terminates the application.

```python
QUIT_BUTTON = "Quit"
```
9.35.25. **RELOAD_LABEL**

Text for the Entry that allows editing of a section reload request.

```plaintext
RELOAD_LABEL = "Reload from"
```

9.35.26. **REMOVE_INSTR_BUTTON**

Text on the button that removes an instructor from the request set.

```plaintext
REMOVE_INSTR_BUTTON = "Remove this instructor from the CMS"
```

9.35.27. **ROLES_FRAME**

Text of the frame around the table of roleholders.

```plaintext
ROLES_FRAME = "Roles"
```

9.35.28. **SAVE_BUTTON**

Text of the button to save the current semester.

```plaintext
SAVE_BUTTON = "Save adds-season-yyyy.xml"
```

9.35.29. **SECTIONS_FRAME**

Title for the frame around the list of sections in the instructor panel.

```plaintext
SECTIONS_FRAME = "Sections: right-click to edit"
```

9.35.30. **SEL_INSTR_FRAME**

Text on the button to select a new instructor.

```plaintext
SEL_INSTR_FRAME = "Click to show an instructor's courses"
```

9.35.31. **SEL_ROLEHOLDER_FINDER**

Text of the UidFinder that searches for the names of potential roleholders.

```plaintext
SEL_ROLEHOLDER_FINDER = "Select roleholder"
```

9.35.32. **SHOW_DEPT_SCHED_FRAME**

Text of the frame around the department selector.

```plaintext
SHOW_DEPT_SCHED_FRAME = "Select department"
```
9.35.33. **TITLE_LABEL**

Text for the Entry that allows editing of a section title.

```
TITLE_LABEL = "Title"
```

9.35.34. **UPLOAD_ACCTS_BUTTON**

Text on the button to run `cmsaccts` and upload the result to the CMS.

```
UPLOAD_ACCTS_BUTTON = "Upload cms accounts"
```

9.35.35. **UPLOAD_COURSES_BUTTON**

Text on the button to run `cmssects` and upload the result to the CMS.

```
UPLOAD_COURSES_BUTTON = "Upload cms courses"
```

9.35.36. **UPLOAD_ENROLL_BUTTON**

Text on the button to run `cmsenroll` and upload the result to the CMS.

```
UPLOAD_ENROLL_BUTTON = "Upload cms enrollments"
```

9.35.37. **ROLE_BUTTON_LIST**

A list of tuples `(label, code)`, where each `label` is a label on a buttons to add a role assignment, and `code` is the corresponding value of the `readadds.Role.kind` attribute.

```
ROLE_BUTTON_LIST = [ 
    (readadds.ROLE_INSTRUCTOR, "Instructor"), 
    (readadds.ROLE_NON_EDITING, "Non-editing teacher"), 
    (readadds.ROLE_GRADER, "Grader")
]
```

9.36. **Popup text constants**

These constants are text that appears on popup menus.

9.36.1. **POP_SECT_TITLE**

Popup title when a section panel isn't valid.

```
POP_SECT_TITLE = "Invalid section title"
```

9.36.2. **POP_DISCARD_SECT**

Button to discard changes when a section panel is invalid.
9.36.3. **POP_RETURN_TO_SECT**
Button to cancel leaving a section panel when it is invalid.

```
POP_RETURN_TO_SECT = "Return to editing this section"
```

9.36.4. **POP_UNSAVED_CHANGES**
Title for the popup when changes haven't been saved.

```
POP_UNSAVED_CHANGES = "You have made unsaved changes"
```

9.36.5. **POP_UNSAVED_CHOICES**
Text for the popup when changes haven't been saved.

```
POP_UNSAVED_CHOICES = "You have not saved your changes to semester {0}."
```

9.36.6. **POP_SAVE_SEMESTER**
Button to save changes on Quit.

```
POP_SAVE_SEMESTER = "Save {0}" "Save {0}""'
```

9.36.7. **POP_DISCARD_SEMESTER**
Button to discard changes to the current semester.

```
POP_DISCARD_SEMESTER = "Discard the changes you have made"
```

9.36.8. **POP_RETURN_TO_SEMESTER**
Button if the user decides not to quit when there are unsaved changes.

```
POP_RETURN_TO_SEMESTER = "Continue editing the {0} semester"
```

9.36.9. **POP_CANCEL**
Cancel saving, as part of the Upload cms courses handler.

```
POP_CANCEL = "Cancel"
```
9.36.10. **POP_INSTR_NONEMPTY**

Title for the popup when deleting an instructor that still has at least one request.

```
POP_INSTR_NONEMPTY = "Instructor has request(s) remaining"
```

9.36.11. **POP_INSTR_REQUEST_COUNT**

Text for popup when deleting a nonempty instructor.

```
POP_INSTR_REQUEST_COUNT = "For instructor {who} ({uid}), section count is {ns},
metacourse count is {nx}""
```

9.36.12. **POP_INSTR_DISCARD**

Button to discard a nonempty instructor.

```
POP_INSTR_DISCARD = "Discard this instructor anyway"
```

9.36.13. **POP_INSTR_CANCEL**

Button to cancel deleting an instructor.

```
POP_INSTR_CANCEL = "Continue editing this instructor"
```

9.36.14. **POP_BAD_META_TITLE**

Title when closing an invalid metacourse.

```
POP_BAD_META_TITLE = "Invalid metacourse title"
```

9.36.15. **POP_BAD_META_TEXT**

Text when closing an invalid metacourse.

```
POP_BAD_META_TEXT = "You must enter a title for this metacourse."
```

9.36.16. **POP_BAD_META_DELETE**

Button to delete an invalid metacourse.

```
POP_BAD_META_DELETE = "Delete this metacourse"
```

9.36.17. **POP_BAD_META_CANCEL**

Button to cancel closing a metacourse.
9.36.18. POP_MISSING_KIDS
Title of the popup when a metacourse doesn't have the minimum number of child sections.

```
POP_MISSING_KIDS = "Missing child section(s)"
```

9.36.19. POP_TWO_KIDS_REQ
Text for the popup when a metacourse doesn't have the minimum number of child sections.

```
POP_TWO_KIDS_REQ = "A metacourse must have two or more child sections."
```

9.36.20. POP_DISCARD_META
Button to discard a flawed metacourse.

```
POP_DISCARD_META = "Discard this metacourse"
```

10. Specification functions

```
# - - - - - S p e c i f i c a t i o n f u n c t i o n s

These functions are notational shorthand for semantic specification in the Cleanroom methodology. Their names always contain multiple words separated by hyphens, so they cannot be confused with objects in Python namespaces.

10.1. adds-file-name: What is the name of the XML file?

The name of the XML file containing instructor course requests is a function of the calendar year and semester name; the semSched argument is an instance of class_sched.SemesterSchedule. See Section 9.5, “ADDS_PREFIX” (p. 26); Section 10.9, “semester-name: Semester code to semester name” (p. 40); Section 10.4, “cal-year: Academic year to calendar year” (p. 39); and Section 9.6, “XML_EXT” (p. 26).

```
# adds-file-name(acadYear, semCode) ==
# ADDS_PREFIX + semester-name(semCode) + "-" +
# cal-year(acadYear, semCode) + XML_EXT
```

Example of such a name for academic year 2019 and semester code 20:

```
adds-fall-2018.xml
```
10.2. arch-dir-name: Archive directory name

This application keeps a number of archive copies of important file in subdirectories of the current directory, one per semester. This specification function gives the name of the archive subdirectory for a given academic year and semester code. See Section 9.7, “ARCH_PREFIX” (p. 26); Section 10.9, “semester-name: Semester code to semester name” (p. 40); and Section 10.4, “cal-year: Academic year to calendar year” (p. 39).

```python
# arch-dir-name(acadYear, semCode) ==
# ARCH_PREFIX + semester-name(semCode).lower() + "-" +
# cal-year(acadYear, semCode) + "/
#
```

10.3. arch-file: Name of an archival backup file

Whenever this application writes a adds-season-yyyy.xml file, it also writes a copy to a subdirectory with this name.

See Section 9.7, “ARCH_PREFIX” (p. 26); Section 10.9, “semester-name: Semester code to semester name” (p. 40); Section 10.4, “cal-year: Academic year to calendar year” (p. 39); and Section 9.5, “ADDS_PREFIX” (p. 26).

```python
# arch-file(acadYear, semCode) ==
# arch-dir-name(acadYear, semCode) +
# ADDS_PREFIX + semester-name(semCode).lower() + "-" +
# cal-year(acadYear, semCode) + "-" +
# (date as yyyy-mm-ddThh-mm-ssZ) + XML_EXT
#
```

Example of such a path:

```
arch-spring-2017/adds-spring-2017-2016-12-28T07:33:42Z
```

10.4. cal-year: Academic year to calendar year

The calendar year of a semester is defined in the class_sched module.

```python
# cal-year(acadYear, semCode) ==
# class_sched.acadYearToCal(acadYear, semCode)
#
```

10.5. default-cal-year: Default calendar year for a given semester

Specifies the default calendar year when a semester was specified but not a calendar year. See Section 9.16, “NEXT_SUMMER_DATE” (p. 28) and Section 9.17, “NEXT_SPRING_DATE” (p. 28).

```python
# default-cal-year(semCode) ==
# if semCode == class_sched.SEM_CODE_SUMMER ->
# if today < NEXT_SUMMER_DATE -> today.yyyy
# else -> str(int(today.yyyy)+1)
# if semCode == class_sched.SEM_CODE_FALL -> today.yyyy
```
# if semCode == class_sched.SEM_CODE_SPRING ->
#   if today < NEXT_SPRING_DATE -> today.yyyy
# else -> str(int(today.yyyy)+1)
#

today.yyyy

10.6. default-cal-year-now: Default calendar year with no semester given

Specifies the rules for selecting a calendar year when no semester was specified on the command line. Starting December 1, we assume the user is working on the following spring.

default-cal-year-now ==
  if today < 12-01 -> today.yyyy
  else -> str(int(today.yyyy)+1)
#

10.7. default-semester-now: Default semester

Specifies the rules for selecting a current semester when none was specified on the command line. See Section 9.18, “SPRING_SUMMER_DATE” (p. 28); Section 9.19, “SUMMER_FALL_DATE” (p. 28); and Section 9.20, “FALL_SPRING_DATE” (p. 28).

default-semester-now ==
  if today < SPRING_SUMMER_DATE -> class_sched.SEM_CODE_SPRING
  else if today < SUMMER_FALL_DATE -> class_sched.SEM_CODE_SUMMER
  else if today < FALL_SPRING_DATE -> class_sched.SEM_CODE_FALL
  else -> class_sched.SEM_CODE_SPRING
#

10.8. prefix-match-count: Number of matching strings

The first argument to cnsadds is a semester name specified as a unique, case-insensitive prefix of one of the standard semester names. This specification function formalizes how we check that the argument is a unique initial substring.

definition (s, nameSet) ==
  the number of elements N of nameSet for which s is a prefix,
  using case-insensitive comparison
#

10.9. semester-name: Semester code to semester name

The conversion of a semester code to a semester name is defined in the class_sched module.

semester-name(semCode) ==
  class_sched.SEM_CODE_TO_NAME[semCode]
10.10. semester-picker-callback: Callback for change of semester

This specification function describes the interface used by the callback described in Section 15, “class SemesterPicker: Widget to show and change the current semester” (p. 88). As discussed in Section 6.1, “Control flow” (p. 22), it’s not always possible to close a transient panel, and changing to a different semester would require closing the current transient panel. So, when the user selects a different semester using the Change semester pulldown menu, the SemesterPicker widget calls its callback function to inform the external world of a request to change semesters. If the callback returns normally, the SemesterPicker displays the new semester’s name. However, if the callback raises the locally defined CloseFailure exception, the widget cancels the change of semester and continues to display whatever semester was shown previously.

```python
# semester-picker-callback ==
# a function c(semSched) with this intended function:
# [ semSched is a class_sched.SemesterSchedule instance ->
#   if a change of semester is permissible ->
#     return None
#   else -> raise CloseFailure ]
```

10.11. std-sem-code: Standardize an abbreviated semester code

Specifies how to translate a prefix of one of the standard semester names into a semester code. The specification states that the given prefix must match one and only one semester name. In the class_sched module, SEM_CODE_TO_NAME is the dictionary that maps semester codes to their standard names; the .values() method on this dictionary returns the set of standard names. This function is undefined unless prefix-match-count(senName) == 1; see Section 10.8, “prefix-match-count: Number of matching strings” (p. 40).

```python
# std-sem-code(senName) ==
# the element of class_sched.SEM_CODE_TO_NAME.values() for which
# senName is a prefix, case-insensitive
#
```

10.12. transient-can-close: Can the transient panel be removed?

This specification function describes a compound condition that results in a failure exception. See Section 6.1, “Control flow” (p. 22) for a discussion of the arbitration of conflicts among different transient panels.

```python
# transient-can-close ==
# (swap area is not displaying a transient) or
# (the transient's contents are in a valid state) or
# (the user selects "discard changes" in a popup)
#
```

10.13. Blanket preconditions

The following preconditions are assumed for the entire application. We will of course test for these failures, but trying to account for some of these relatively uncommon failures tends to complicate the construction of intended functions to a needless extent.
11. Main program: main()

The first step is to check the command line arguments, which are encapsulated as a singleton `Args()` instance; see Section 12, “class Args(): Command line argument singleton” (p. 43). The rest is brief and pro forma: it instantiates the application class (see Section 14, “class App: The application as a whole” (p. 48)), titles its window, and enters its event loop.

```python
# - - - - - main

def main():
    '''Main program.'''
    #-- 1
    # [ if the command line arguments are valid ->
    #     Args() := a singleton Args instance representing them
    # else ->
    #     sys.stderr := error message
    #     stop execution ]
    Args()

Next we read the class schedule file through its Python interface.

```
Check that there is schedule information for the semester specified on the command line.

```python
#-- 3
# [ if classSchedule has data for the semester with academic year
#   Args().acadYear and semester code Args().semCode ->
#   I
#   else ->
#   sys.stderr +:= error message
#   stop execution ]
try:
    semSched = classSchedule.lookupSemester(Args().acadYear,
                                           Args().semCode)
except KeyError:
    fatal("*** The current class schedule has no data for the "
          "{0} semester.".format(
          class_sched.semesterName(Args().acadYear,
                                   Args().semCode)))
```

Now start the application.

```python
#-- 4
# [ (display) +:= an instance of App, using classSchedule for
#   scheduling information and semSched for the initially
#   selected semester, and responding to events ]
app = App(classSchedule, semSched)
app.master.title("{0} {1}".format(PROGRAM_NAME, VERSION))
app.mainloop()
```

### 12. `class Args()`: Command line argument singleton

This singleton processes the command line arguments. The class variable `INITIALIZED` is initially `False`; it is set to `True` once all the arguments are processed. Thus, subsequent references do not reprocess the arguments.

Only the academic year and semester code are exported. The calendar year (`_calYear`) is used internally.

```python
# - - - - - class Args

class Args(singleton.Singleton):
    '''Encapsulates the command line arguments.

    Exports:
    Args():
        [ if this class has already been instantiated ->
          return the singleton instance
        else if the command line arguments are valid ->
          return a singleton instance representing those
          arguments
        else ->
          sys.stderr +:= error message(s)
```
12.1. Args.__init__()

```python
def __init__(self):
    '''Constructor.'''

    #-- 1
    if Args.INITIALIZED:
        return
    else:
        Args.INITIALIZED = True
```

The first step is to see how many arguments there are and syntax-check them. The end result of this process is to set the `.semCode` and `.calYear` attributes to their specified values, or to `None` if omitted. See Section 12.2, “Args._syntaxCheck(): Syntax-check the command line” (p. 45).

```python
#-- 2
# [ if the command line is syntactically correct ->
#   self.semCode := std-sem-code(semester argument),
#   or None if omitted
#   self._calYear := the specified calendar year as a
#   four-digit string, or None if omitted ]
self._syntaxCheck()
```

Now we apply the rules for default values. See Section 12.5, “Args._applyDefaults(): Set defaults for year and semester code” (p. 47).

```python
#-- 3
# [ if self.semCode is None ->
#   self.semCode := default-semester-now
#   self._calYear := default-cal-year-now
# else if self._calYear is None ->
#   self._calYear := default-cal-year(self.semCode)
# else -> I ]
self._applyDefaults()
```

At this point we know that `.semCode` and `.calYear` are correctly set. The last step is to compute the academic year.
```python
#-- 4
# \[ self.acadYear := acad-year(self.semCode, self._calYear) \] 
if self.semCode == class_sched.SEM_CODE_SPRING:
    self.acadYear = self._calYear 
else:
    self.acadYear = str(int(self._calYear)+1)
#-- 5
self.semName = class_sched.SEM_CODE_TO_NAME[self.semCode]
```

### 12.2. Args._syntaxCheck(): Syntax-check the command line

```python
#-- 1
# \[ argParser := an argparse.ArgumentParser instance \]
argParser = argparse.ArgumentParser( 
    description="CMS course import manager")
```

The first step is to build an argument parser.

```python
#-- 2
# \[ argParser +:= a single, optional positional argument \n# \[ that stores std-sem-code(its argument) under .semCode, \n# \[ defaulting to None \]
argParser.add_argument("semCode", metavar='SEMMESTER’, 
    nargs='?', default=None, 
    type=self._checkSemester, 
    help="Semester name: Fall, Spring, Summer or a unique " 
    "prefix (case-insensitive)")
```

For the function that checks the semester name and converts it to a semester code, see Section 12.3, “Args._checkSemester(): Validate the semester name” (p. 46).

```python
#-- 3
# \[ argParser +:= a single, optional positional argument \n# \[ that stores its argument under ._calYear, defaulting to 
```

For the function that checks the calendar year, see Section 12.4, “Args._checkYear(): Validate the calendar year” (p. 47).
argParser.add_argument("_calYear", metavar='YYYY',
nargs='?', default=None,
type=self._checkYear,
help="Four-digit calendar year")

By using namespace=self here, we ask the argument parser to store the values in the Args singleton.

cmsadds

argParser.parse_args(namespace=self)

12.3. Args._checkSemester(): Validate the semester name

def _checkSemester(self, s):
    'Check the semester name and convert to a semester code.

    [ s is a str ->
      if prefix-match-count(s,
        class_sched.SEM_NAME_TO_CODE.keys()) == 1 ->
        return std-sem-code(semName)
      else -> raise argparse.ArgumentTypeError ]

First we form the set of semester names for which \( s \) is a prefix, case-insensitive.

cmsadds

uc = s.upper()

matches = [ name
    for name in class_sched.SEM_NAME_TO_CODE.keys()
    if name.upper().startswith(uc) ]

The author had to think for a bit about how to retrieve the value from a singleton set. Solution: convert it to a list and take the first element. (Another way is to use an assignment such as "name, = matches").

cmsadds

match = list(matches)[0]
12.4. Args._checkYear(): Validate the calendar year

```python
# -- -- A r g s . _ c h e c k Y e a r

YEAR_PATTERN = re.compile(  
    r'20'  # Must start with 20; that'll take us to 2100
    r'\d{2}'  # Matches two more digits
    r'$')  # Insure a complete match

def _checkYear(self, s):
    '''Check the calendar year for syntax.

    [ if s matches self.YEAR_PATTERN ->
      return s
    else -> raise argparse.ArgumentTypeError ]
    '''
    #-- 1
    m = Args.YEAR_PATTERN.match(s)
    if m is not None:
        return s
    else:
        raise argparse.ArgumentTypeError("Year must be in "
                                          "the range 2000-2099.")
```

12.5. Args._applyDefaults(): Set defaults for year and semester code

```python
# -- -- A r g s . _ a p p l y D e f a u l t s

def _applyDefaults(self):
    '''Set default year and semester if omitted.

    [ if self.semCode is None ->
      self.semCode := default-semester-now
      self._calYear := default-cal-year-now
    else if self._calYear is None ->
      self._calYear := default-cal-year(self.semCode)
    else -> I ]
    '''
    #-- 1
    # [ monthDay := a tuple (today's month as an int, today's
    #   day of the month as an int)
    # thisYear := this year as a four-digit str
    # nextYear := next year as a four-digit str ]
    today = datetime.date.today()
    monthDay = (today.month, today.day)
```
thisYear = str(today.year)
nextYear = str(today.year+1)

See: Section 10.7, “default-semester-now: Default semester” (p. 40); Section 10.6, “default-calyear-now: Default calendar year with no semester given” (p. 40); and Section 10.5, “default-calyear: Default calendar year for a given semester” (p. 39).

```python
#-- 2
if self.semCode is None:
    if monthDay < SPRING_SUMMER_DATE:
        self.semCode = class_sched.SEM_CODE_SPRING
        self._calYear = thisYear
    elif monthDay < SUMMER_FALL_DATE:
        self.semCode = class_sched.SEM_CODE_SUMMER
        self._calYear = thisYear
    elif monthDay < FALL_SPRING_DATE:
        self.semCode = class_sched.SEM_CODE_FALL
        self._calYear = thisYear
    else:
        self.semCode = class_sched.SEM_CODE_SPRING
        self._calYear = nextYear
elif self._calYear is None:
    if self.semCode == class_sched.SEM_CODE_SUMMER:
        if monthDay < NEXT_SUMMER_DATE:
            self._calYear = thisYear
        else:
            self._calYear = nextYear
    elif self.semCode == class_sched.SEM_CODE_FALL:
        self._calYear = thisYear
    elif monthDay < NEXT_SPRING_DATE:
        self._calYear = thisYear
    else:
        self._calYear = nextYear
```

13. Locally defined exceptions

When the user tries to close a transient panel in the swap area, and that panel is not in a valid state, and the user elects to stay in that panel and fix it, the application logic raises this exception. It indicates to the caller that whatever operation needed that panel closed has failed and should not continue.

```python
#-- - c l a s s C l o s e F a i l u r e

class CloseFailure(Exception):
    '''Exception class when the transient panel can't be closed.
    '''
    pass
```

14. class App: The application as a whole

The application inherits from ttk.Frame.
class App(ttk.Frame):
    '''The application as a whole.

Exports:
    App(classSchedule, semSched):
        [ (classSchedule is an instance of
class sched.ClassSchedule) and
        (semSched is a class_sched.SemesterSchedule in
classSchedule) ->
        (display) +:= an instance of self using classSchedule
        for scheduling information and initially displaying
        semSched
    .classSchedule: [ as passed to constructor, read-only ]
    .semSched:
        [ if self displays no semester -> None
        else ->
        the current semester as a class_sched.SemesterSchedule ]
    .lookupDeptName(deptCode):
        [ deptCode is a str ->
        if deptCode is defined in self ->
        return the corresponding department name
        else -> raise KeyError ]
    .lookupCRN(crn):
        [ if there is a request for a section with CRN=(crn) in
        self's current semester ->
        return the readadds.Section instance for that section
        else -> raise KeyError ]
    .lookupCRNSched(crn):
        [ if there is a section with CRN=(crn) in self's current
        semester schedule ->
        return the class_sched.SectionSchedule instance for
        that section
        else -> raise KeyError ]

Here are the operational methods that pass control flow around the application. See Section 6.1, “Control flow” (p. 22) and Section 10.12, “transient-can-close: Can the transient panel be removed?” (p. 41) for a discussion of the management of transients in the swap area.

The .close() method closes the transient (if there is one) and the ReqPanel, and saves the current request set (unless the user elects to discard the changes).

    .close():
        [ if not transient-can-close ->
        raise CloseFailure
        else if (self. addsFile is None) or
        (self. addsFile has no unsaved changes) or
        (the user selects "Discard" in a "Save changes?" popup) ->
        self := self with a blank ReqPanel and no transient
        else if file adds-file-name(self.semSched) is writeable ->
self := self with a blank ReqPanel and no transient
that file := self._addsFile as XML
self._addsFile := self._addsFile marked clean
else ->
    self := self with a blank ReqPanel and no transient
    raise CloseFailure ]

The .closeTransient() method just closes the transient panel.

cansadds

.closeTransient():
    [ if transient-can-close ->
    self := self displaying no transient
    else -> raise CloseFailure ]

The .openSemester() method is called to change to a different semester.

cansadds

.openSemester(newSched):
    [ newSched is a class_sched.SemesterSchedule instance ->
    if file adds-file-name(newSched) does not exist ->
    self := self displaying semester (newSched)
    self._addsFile := a new readadds.AddsFile instance
    for semester (newSched)
    else if file adds-file-name(newSched) is readable and
    valid against SCHEMA_FILE ->
    self := self displaying semester (newSched)
    self._addsFile := a readadds.AddsFile instance
    representing that file
    else -> raise CloseFailure ]

The next few methods implement the operations discussed in Section 6.1, “Control flow” (p. 22).

cansadds

.reqBlank():
    [ self := self with the ReqPanel blanked out ]
.reqSelect(teacher):
    [ teacher is a readadds.Teacher instance ->
    self := self with the ReqPanel displaying teacher ]
.removeInstructor(uid):
    [ uid is a TCC user ID ->
    self := self without the req element for the
    instructor whose user ID is (uid) ]
.showDept(deptCode):
    [ (self.semSched is a class_sched.SemesterSchedule) and
    (deptCode is a str) ->
    if deptCode is a department code in the schedule
    for self.semSched ->
    if transient-can-close ->
    self := self displaying the SchedPanel for
    the department with code (deptCode)
    else -> raise CloseFailure
    else -> (display) += error popup: no classes scheduled
    in dept (deptCode) for semester self.semSched ]
.showSection(section):
[ (self is displaying an instructor) and
  (section is a readadds.Section currently in that
  instructor's request set) ->
  if transient-can-close ->
    self := self displaying the SectionPanel for (section)
  else -> I ]

.showMeta(crossParent):
  [ crossParent is a readadds.CrossParent instance ->
    if transient-can-close ->
      self := self displaying the MetaPanel for
        editing (crossParent)
    else -> raise CloseFailure ]

.deleteMeta(crossParent):
  [ (self._reqPanel is a ReqPanel instance) and
    (crossParent is a readadds.CrossParent in self._reqPanel) ->
    self._reqPanel := self._reqPanel with crossParent
      removed
    self := self displaying no transient ]

.getSelectedSection():
  [ if (self shows a ReqPanel) and
    (a section in that ReqPanel is highlighted) ->
    return that section's CRN
  else -> raise ValueError ]

.clearSpecificMeta(crossParent):
  [ crossParent is a readadds.CrossParent ->
    if (self._swapArea contains self._metaPanel) and
      (self._metaPanel is currently displaying crossParent ->
      self._metaPanel := self._metaPanel cleared
      self._swapArea := self._swapArea cleared
    else -> I ]

.updateReqPanel():
  [ if self._reqPager is displaying self._reqPanel ->
    self._reqPanel := self._reqPanel with its lists of
      sections and metacourses rebuilt from its
      current teacher
  else -> I ]

Class internals include the readadds.AddsFile instance that holds the set of current requests.

State/Invariants:
  .addsFile:
    [ if a semester has been selected ->
      a readadds.AddsFile representing a set of course
      requests
    else -> None ]

The next group of internal attributes are the child widgets and their associated control variables.

Widgets and control variables:
  .mainFrame: [ encloses all general controls ]
  .teacherFinder:
    [ the uidfinder.UidFinder instance for selecting a primary
      instructor ]
Here are the six main control groupings that are managed by the `App` class. The `._mainFrame` encloses the area on the top of the application with controls that are always visible. The `._reqFrame` attribute is a frame that holds the `ReqPanel`; its value is `None` when the `ReqPanel` is closed. The swap area is `self._swapArea`, and the last three are its transient child panels.
14.1. App.semSched: Current semester schedule

This attribute is actually a read-only property that passes through the .semSched attribute of the SemesterPicker.

```python
# - - - A p p . s e m S c h e d
def semSched(self):
    '''Current SemesterSchedule'''
    return self._semPicker.semSched
```

14.2. App.__init__()  

First we call our parent constructor and save the schedule data.
In Section 14.3, “App._styler(): Set up fonts and ttk styles” (p. 54), we modify the ttk widget styles of its standard widgets, create some variant styles, and define fonts for the non-ttk widgets (such as tk.Listbox).

All the widgets are created in Section 14.4, “App._createWidgets()” (p. 55).

The final step is to try to open the adds-season-yyyy.xml. If that fails, we create a new empty AddsFile instance for the given semester.

14.3. App._styler(): Set up fonts and ttk styles

Most of the widgets we use come from the ttk widget set, which use the concept of named styles to set up many of their visual attributes. This method configures several of the canned styles and also creates derived styles, such as labels with a font. For the non-ttk widgets we use (such as tk.Listbox), the constructors can use the fonts we create here.
def _styler(self):
    '''Set up fonts and ttk styles.
    '''
    #-- 1
    self._baseFont = tkFont.Font(family=BASE_FONT_FAMILY, size=14)
    self._titleFont = tkFont.Font(family=BASE_FONT_FAMILY, size=20)
    self._smallFont = tkFont.Font(family=BASE_FONT_FAMILY, size=12)
    self._entryFont = tkFont.Font(family=MONOSPACE_FAMILY, size=24)

    #-- 2
    baseStyle = ttk.Style()
    baseStyle.configure('.',
                       foreground=MAIN_FG_COLOR,
                       background=MAIN_BG_COLOR,
                       activebackground=ACTIVE_BG_COLOR,
                       font=self._baseFont)

    #-- 3
    panelTitleStyle = ttk.Style()
    panelTitleStyle.configure(PANEL_TITLE_STYLE, font=self._titleFont)

    #-- 4
    entryStyle = ttk.Style()
    entryStyle.configure(ENTRY_STYLE, font=self._entryFont)

    #-- 5
    frameStyle = ttk.Style()
    frameStyle.configure(FRAME_STYLE,
                         height=100, width=100, padx=10)

    #-- 6
    labelFrameStyle = ttk.Style()
    labelFrameStyle.configure(LABEL_FRAME_STYLE,
                               font=self._baseFont)

14.4. App._createWidgets()

We divide widget creation into three areas corresponding to the three main divisions of the top-level grid. See Section 14.7, “App._mainWidgets(): Create the widgets in the general controls panel” (p. 59); Section 14.21, “App._reqWidgets(): Create widgets in the primary instructor panel” (p. 72); and Section 14.22, “App._swapWidgets(): Create the swap area and its children” (p. 73).
### 14.5. `App._selInstrHandler()`: Bring up one instructor’s course list

This is the handler called when the user clicks on a name in the listbox inside the Click to show an instructor’s courses frame. The `tccAccount` argument is an instance of Section 32.19, “`uidfinder.py: class TccAccount`” (p. 192). Here is the flow:

1. Before we can display the new instructor, we must remove the old instructor from the `ReqPanel`, including any transient panel that was invoked from the old instructor. This is done by calling `self._reqPanel.close()`; if the close fails, this method will raise a `CloseFailure` exception, in which case we return without changing to the new instructor.

2. Next we determine whether the current `AddsFile` has a `Teacher` instance for this instructor’s UID. If so, we displaying their existing sections and metacourses.

3. If the `AddsFile` has no entry, we create a new `Teacher` instance and attach it to the `AddsFile`.

4. Using the `Teacher` instance from the previous step, call `self._reqPanel.set()` to display that instance.
[ if not transient-can-close ->
  raise CloseFailure
else if self._addsFile has a Teacher instance whose email matches tccAccount.uid ->
  self._reqPanel := self._reqPanel displaying that Teacher instance
  self := self showing no transient
else ->
  self._addsFile +:= a new Teacher instance made from tccAccount
  self.semSched +:= that Teacher instance
  self._reqPanel := self._reqPanel displaying that Teacher instance
  self := self showing no transient ]

See Section 16, “class ReqPanel: The primary instructor edit panel” (p. 92).

#-- 1
# [ if (transient-can-close) and
# (self._reqPanel is in a valid state) ->
#   self := self showing no ReqPanel or transient
# else -> raise CloseFailure ]
self._reqPanel.close()

Now see if self._addsFile has an entry for this instructor. If there isn’t one, we must fabricate one and add it to the tree; the logic that does this is Section 14.6, “App._newTeacher(): Create a new readadds.Teacher from a uidfinder.TccAccount” (p. 58).

#-- 2
# [ if self._addsFile has a readadds.Teacher instance
#   with email==tccAccount.uid ->
#     teacher := that readadds.Teacher instance
#   else ->
#     self._addsFile +:= a new Teacher instance made from tccAccount
#     teacher := that Teacher instance ]
try:
  teacher = self._addsFile.getTeacher(tccAccount.uid)
except KeyError:
  teacher = self._newTeacher(tccAccount)

#-- 3
# [ self._reqPanel := self._reqPanel editing teacher
#   self._reqPager := self._reqPager showing self._reqPanel ]
self._reqPanel.select(teacher)
sel._reqPager.select(self._reqPanel)
14.6. App._newTeacher(): Create a new readadds.Teacher from a uidfinder.TccAccount

The fundamental shortcoming of the current TCC LDAP directory is that it does not separate names into given, middle, and family names, but the purpose of this method is create a readadds.Teacher instance, which does require that distinction.

Accordingly, we will try to guess, and leave the failures for the user to clean up. The intent is that when an instructor appears in the ReqPanel, and the user notices that the names aren't right, they will edit them, referring to the official directories if necessary.

So our first-approximation technique is this:

• Some people, e.g., from Indonesia, have only one name, so if the TCC name has only one word, we'll put it in the family name field (since that is used to alphabetize names) and leave the other two empty.
• If the TCC name has two words, we'll assume the first is the given name and the second is the family name.
• If the TCC name has three or more words, we'll use the first as a given name, the second as a middle name, and the rest as the family name.

```python
# - - - A p p . _ n e w T e a c h e r

def _newTeacher(self, tccAccount):
    '''Add a new Teacher instance to the current AddsFile.

    [ tccAccount is a uidfinder.TccAccount instance ->
      self._addsFile += a new readadds.Teacher instance
      made from tccAccount
      return that Teacher instance ]
    '''
    #-- 1
    # [ wordList := a list of the space-separated words in
tccAccount.fullName
    #   given := ''
    #   middle := ''
    #   family := 'UNKNOWN' ]
    wordList = tccAccount.fullName.split()
given = ''
middle = ''
family = 'UNKNOWN'

    #-- 2
    if len(wordList) == 1:
        family = wordList[0]
    elif len(wordList) == 2:
        given = wordList[0]
        family = wordList[1]
    else:
        given = wordList[0]
        middle = wordList[1]
        family = ' '.join(wordList[2:])

    #-- 3
```
# [ self._addsList := self._addsList with a new Teacher
# instance made from email=tccAccount.uid, given=(given),
# middle=(middle), and family=(family)
# teacher := that Teacher instance ]

# -- 4
#
# return teacher

14.7. **App._mainWidgets(): Create the widgets in the general controls panel**

For the grid plan, see Section 14, “class App: The application as a whole” (p. 48).

```python
# - - - A p p . _ m a i n W i d g e t s

def _mainWidgets(self):
    '''Create widgets and variables in the general controls area.
    
    f = ttk.Frame(self)
    rowx, colx = 0,0
    self._mainLabel = ttk.Label(f,
        text='cmsadds: CMS course request manager',
        style=PANEL_TITLE_STYLE)
    self._mainLabel.grid(row=rowx, column=colx, columnspan=2)

    The Click to show an instructor's courses group is an instance of the UidFinder widget; see Section 32, “Module uidfinder.py” (p. 176). For the handler that is invoked when the user clicks on a name in the list, see Section 14.5, “App._selInstrHandler(): Bring up one instructor's course list” (p. 56).

    rowx, colx = rowx+1, 0
    self._teacherFinder = uidfinder.UidFinder(f,
        "Click to show an instructor's courses",
        callback=self._selInstrHandler)
    self._teacherFinder.grid(row=rowx, column=colx,
        padx=5, pady=5)

    Frame self._mainControls holds the semester selector and various buttons; see Section 14.8, “App._createControls(): Create general controls” (p. 59).

    colx += 1
    self._mainControls = self._createControls(f)
    self._mainControls.grid(row=rowx, column=colx,
        padx=5, pady=5, sticky=tk.N+tk.W)

    return f
```

14.8. **App._createControls(): Create general controls**

This method creates and returns a `ttk.Frame` containing the semester picker and various buttons.
# A p p . _ c r e a t e C o n t r o l s

def _createControls(self, f):
    '''Create self._mainControls and the widgets inside it.
    ...'''
    controls = ttk.Frame(f)

The widget group for displaying and changing the current semester is an instance of Section 15, “class SemesterPicker: Widget to show and change the current semester” (p. 88). For the callback, see Section 14.9, “App._semChangeHandler() : Handler for changing semesters” (p. 61).

```python
rowx = 0
self._semPicker = SemesterPicker(controls, self.classSchedule,
                                 self._semChangeHandler)
self._semPicker.grid(row=rowx)
```

For the logic behind the Save adds-season-yyy.xml button, see Section 14.10, “App._saveHandler() : The Save adds-season-yyy.xml button” (p. 62).

```python
rowx += 1
self._saveTitle = tk.StringVar()
self._saveTitle.set('Save adds-season-yyy.xml')
self._saveButton = ttk.Button(controls,
                              textvariable=self._saveTitle,
                              command=self._saveHandler)
self._saveButton.grid(row=rowx, pady=5)
```

For the Upload cms accounts button, see Section 14.11, “App._uploadAcctsHandler() : The Upload cms accounts button” (p. 62).

```python
rowx += 1
self._uploadAcctsButton = ttk.Button(controls,
                                      text='Upload cms accounts',
                                      command=self._uploadAcctsHandler)
self._uploadAcctsButton.grid(row=rowx, pady=5)
```

For the Upload cms courses button, see Section 14.16, “App._uploadCoursesHandler() : The Upload cms courses button” (p. 67).

```python
rowx += 1
self._uploadCoursesButton = ttk.Button(controls,
                                        text='Upload cms courses',
                                        command=self._uploadCoursesHandler)
self._uploadCoursesButton.grid(row=rowx, pady=5)
```

For the Upload cms enrollments button, see Section 14.17, “App._uploadEnrollHandler() : The Upload cms enrollments button” (p. 68).

```python
rowx += 1
self._uploadEnrollButton = ttk.Button(controls,
                                      text='Upload cms enrollments',
                                      command=self._uploadEnrollHandler)
self._uploadEnrollButton.grid(row=rowx, pady=5)
```
For the Cleanup *cms drops* button, see Section 14.18, “App._cleanupDropsHandler(): The Cleanup cms drops button” (p. 69).

```python
rowx += 1
def _cleanupDropsButton(self, controls, text='Cleanup cms drops',
                       command=self._cleanupDropsHandler)
    self._cleanupDropsButton.grid(row=rowx, pady=5)
```

See Section 14.20, “App._quitHandler(): The Quit button” (p. 71). We can't just quit; there may be an invalid transient—see Section 6.1, “Control flow” (p. 22).

```python
rowx += 1
def _quitButton(self, controls, text=QUIT_BUTTON,
                command=self._quitHandler)
    self._quitButton.grid(row=rowx, pady=5)
```

**14.9. App._semChangeHandler(): Handler for changing semesters**

Before we can change to a different semester, we must close any transient (if possible) and save changes (if necessary). See Section 14.27, “App.close(): Remove the current semester” (p. 75) and Section 14.30, “App.openSemester(): Switch to a different semester” (p. 78).

```python
# - - - A p p . _ s e m C h a n g e H a n d l e r
def _semChangeHandler(self, newSched):
    '''Try to change to a different semester.
    '''
    #-- 1
    # [ if not transient-can-close ->
    # return
    # else if (self._addsFile is None) or
    # (self._addsFile has no unsaved changes) or
    # (the user selects "Discard" in a “Save changes?”
    # self := self with a blank ReqPanel and no transient
    # else if file adds-file-name(self.semSched) is writeable ->
    # self := self with a blank ReqPanel and no transient
    # that file := self._addsFile as XML
    # self._addsFile := self._addsFile marked clean
    # else ->
    # self := self with a blank ReqPanel and no transient
    # return ]
    try:
        self.close()
    except CloseFailure:
        return
```
14.10. App._saveHandler(): The Save adds-season-yyvy.xml button

For the derivation of the name of the save file, see Section 28, “addsFileName(): The name of the save file” (p. 175).

```python
#-- 2
# [ if file adds-file-name(newSched) does not exist ->
#   self := self displaying semester (newSched)
#   self._addsFile := a new readadds.AddsFile instance
#     for semester (newSched)
# else if file adds-file-name(newSched) is readable and
#   valid against SCHEMA_FILE ->
#   self := self displaying semester (newSched)
#   self._addsFile := a readadds.AddsFile instance
#     representing that file
# else -> return ]
try:
  self.openSemester(newSched)
except CloseFailure:
  return
```

14.11. App._uploadAcctsHandler(): The Upload cms accounts button

This handler runs cmsaccts and uploads the result. For the actual command that will be run, see Section 9.10, “CMSACCTS_CMD” (p. 27); execution and uploading are handled by Section 14.12, “App._imsUpload(): Build and upload an IMS file into the CMS” (p. 63).

```python
#-- 1
try:
  self._saver()
except CloseFailure as x:
  pass
```
14.12. App._imsUpload(): Build and upload an IMS file into the CMS

This method is used by the various upload buttons such as Upload cms accounts to run one of the applications in the cmsimport suite that generates an IMS Enterprise XML file, moves it to the appropriate archive directory, and then uploads it to the CMS.

Each of the applications (cmsaccts, cmssects, cmsenroll, and mooroster) writes a line showing the name of the generated file. This line is written by the Enterprise.write() method in the cmsimport suite. Here is an example of that output line from cmssects:

```
Output file is 'sects_2015-01-16T01-35-54Z.xml'.
```

This method searches the output from the application for a line that looks like that, using a regular expression: see Section 9.14, “OUT_FILE_LINE_PAT” (p. 27).

The file is always written to the current working directory, but we will move it to the appropriate semester's archive directory before uploading it to the CMS.
First, run the command and capture its output. See Section 14.13, “App._runImsCommand(): Build an IMS file” (p. 64).

```python
#-- 1
# [ if (command) is successful ->
#   current directory +:= IMS file created by (command)
#   imsName := name of that IMS file
# else -> raise CloseFailure ]
imsName = self._runImsCommand(command)
```

Next, move the IMS file to the current archive directory. See Section 14.14, “App._imsArchive(): Move the IMS file to its archive directory” (p. 65).

```python
#-- 2
# [ if arch-dir exists or can be created ->
#   arch-dir +:= file (imsName)
#   newPath := name of file (imsName) in arch-dir
# else -> raise CloseFailure ]
newPath = self._imsArchive(imsName)
```

Finally, upload the IMS file to the CMS server. See Section 14.15, “App._copyCms(): Transfer an IMS Enterprise file to the CMS server” (p. 66).

```python
#-- 3
# [ if user is authorized to upload to Moodle ->
#   (Moodle server) +:= file (newPath)
# else -> raise CloseFailure ]
self._copyCms(newPath)
```

### 14.13. App._runImsCommand(): Build an IMS file

See the note in Section 14.12, “App._imsUpload(): Build and upload an IMS file into the CMS” (p. 63) discussing how we find out the name of the file that was written, and see Section 9.14, “OUT_FILE_LINE_PAT” (p. 27) for the regular expression we use.

**Note**

The timeout argument to pexpect.run() is most important! The stock timeout is 30 seconds, and moor roster will typically take more time than that. The symptom is that the output is truncated, and missing the line that identifies the output file name.
[ command is a command that creates an IMS file in the current directory using cmsload.Enterprise.write() ->
  if (command) is successful ->
    current directory := result of (command)
    return the name of the file built by (command)
  else -> raise CloseFailure ]

#-- 1
# [ results := standard output from executing (command) ]
results = pexpect.run(command, timeout=10000)

#-- 2
# [ if results contains a line that matches OUT_FILE_LINE_PAT ->
#   return group 1 from the first such match
#   else -> I ]
for resultLine in results.split('
'):
  m = OUT_FILE_LINE_PAT.match(resultLine)
  if m is not None:
    imsName = m.group(1)
    return imsName

#-- 3
raise CloseFailure("Command: {0}\nResults:\n{1}".format(command, results))

14.14. App._imsArchive(): Move the IMS file to its archive directory

```python
# - - - A p p . _ i m s A r c h i v e
def _imsArchive(self, imsName):
  '''Move an IMS file to the current archive directory.

  [ let
    arch-dir == arch-dir-name(self.semSched.acadYear,
                              self.semSched.semCode)
  in ->
    imsName names a file ->
      if arch-dir exists ->
        directory (arch-dir) ::= file (imsName)
        return the path to (imsName) within arch-dir
      else if arch-dir can be created ->
        directory(arch-dir) := a new subdirectory containing
        file (imsName)
        return the path to (imsName) within arch-dir
      else -> raise CloseFailure ]
  ...
```

See Section 14.33, “App._archDirName(): Archive directory name” (p. 81).

```python
#-- 1
# [ archDirName := arch-dir ]
archDirName = self._archDirName(self.semSched.acadYear,
```
See Section 14.35, “App._makeArchDir(): Build the archive directory if necessary” (p. 82).

14.15. App._copyCms(): Transfer an IMS Enterprise file to the CMS server

For the command that performs the actual CMS upload, see Section 9.15, “MOODLE_UPLOAD_CMD” (p. 28).

```python
def _copyCms(self, imsPath):
    '''Transfer an IMS Enterprise file to the CMS server.

    [ imsPath names a readable file ->
      if the user is authorized to upload to Moodle ->
        (Moodle server) +=: file (imsPath)
      else -> raise CloseFailure ]
    '''
    command = MOODLE_UPLOAD_CMD.format(imsPath)
    result = os.system(command)
    if result != 0:
        raise CloseFailure("Upload failure {0}".format(result))
```

---

```python
newName = os.path.join(archDirName, imsName)

#-- 3
# [ if the directory containing newName exists ->
#   I
# else if that directory can be created ->
#   (file system) +=: that directory
# else -> raise CloseFailure ]
self._makeArchDir(newName)

#-- 4
# [ if file (imsName) can be moved to file (newName) ->
#   file (newName) := file(imsName)
# else -> raise CloseFailure ]
try:
    os.rename(imsName, newName)
except (IOError, OSError) as x:
    raise CloseFailure("Can't rename '{0}' as '{1}': {2}".format(imsName, newName, str(x)))

#-- 5
return newName
```
14.16. App._uploadCoursesHandler(): The Upload cms courses button

Before we can run cmssects to generate an IMS file that creates the courses, we must first insure that all current changes are saved to the adds-season-yyyy.xml file.

For the logic that runs cmssects and uploads it to the CMS, see Section 14.12, “App._imsUpload(): Build and upload an IMS file into the CMS” (p. 63).

Note the precondition “self._addsFile is a readadds.AddsFile instance”. As initially designed, by the time the application goes into its .mainloop(), a semester is selected and self._addsFile is set to the corresponding new or reloaded readadds.AddsFile instance.

---

```python
def _uploadCoursesHandler(self):
    '''Handler for the 'Upload cms courses' button.

    [ let
        save-file == file adds-file-name(self.semSched.acadYear,
        self.semSched.semCode)
        save-data == self._addsFile serialized as XML
        arch-dir == directory arch-dir-name(
        self.semSched.acadYear, self.semSched.semCode)
    in ->
        self._addsFile is a readadds.AddsFile instance ->
            if self._addsFile has unsaved changes ->
                if the user selects "Cancel" in a "Save/Cancel"
                popup ->
                    I
                else ->
                    save-file := save-data
                    arch-dir +=: (a timestamped copy of save-data) +
                    cmssects(save-data)
                    (Moodle) +=: cmssects(save-data)
            else ->
                (file system) := cmssects(save-data)
                arch-dir +=: (a timestamped copy of save-data) +
                cmssects(save-data)
                (Moodle) +=: cmssects(save-data) ]
```
14.17. App._uploadEnrollHandler(): The Upload cms enrollments button

This handler runs cmsenroll and uploads the result. However, first we must insure that changes have been saved to the adds-season-yyyy.xml file; see Section 14.19, “App._saveOrCancel(): Insure changes have been saved” (p. 70).

def _uploadEnrollHandler(self):
    '''Handler for the "Upload cms enrollments" button
    '''
    #-- 1
    # [ if self.addsFile has unsaved changes ->
    #   if the user selects "Cancel" in a "Save/Cancel" popup ) ->
    #     return
    #   else ->
    #     save-file := save-data
    #     arch-dir +=: a timestamped copy of save-data
    #   else -> I ]
    reply = self._saveOrCancel()
    if reply == POP_CANCEL:
        return
    
    #-- 2
    # [ inFileName := adds-file-name(self.semSched.acadYear,
    #    self.semSched.semCode) ]
    inFileName = addsFileName(self.semSched)
# [ if (running CMSENROLL_CMD.format(inFileName) produces
# an IMS file) and
# (the user is authorized to upload to Moodle) ->
# arch-dir +=: that IMS file
# (Moodle) +=: that IMS file
# else ->
# (display) +=: an error popup ]
try:
    command = CMSENROLL_CMD.format(inFileName)
    self._imsUpload(command)
except CloseFailure as x:
    title = "cmsenroll failure"
    text = str(x)
    tkMessageBox.showerror(title, text)

def _cleanupDropsHandler(self):
    '''Handler for the Cleanup cms drops button.

    [ let
        arch-dir == arch-dir-name(self.semSched.acadYear,
                                  self.semSched.semCode))
    in ->
        if (the user is authorized to run mooroster) and
        (the Banweb interface is available) and
        (the user is authorized to upload to the CMS) ->
        (Moodle) +=: an IMS Enterprise file that deletes
        Moodle enrollments not in Banner
        directory (arch-dir) +=: that IMS Enterprise file
        else -> raise CloseFailure ]

14.18. App._cleanupDropsHandler(): The Cleanup cms drops button

The purpose of the Cleanup cms drops button is to run the mooroster application with the options that
ask it to compare Moodle’s current enrollments with Banner’s class rosters and generate an IMS Enterprise
file that disenrolls students who are in Moodle but no longer in Banner. This file’s name will start with
“cleanup...”. The actual command and its options are defined in Section 9.13,
“MOOROSTER_CMD” (p. 27); execution and uploading are handled by Section 14.12, “App._imsUpload(): Build and upload an IMS file into the CMS” (p. 63).

The mooroster application does not depend on the currently selected semester: it always operates on
the current Moodle semester. However, we also save the resulting cleanup... file in the archive
directory of the current semester, just so we have an audit trail showing what files were uploaded when.

Note

This function contends for a unique resource, the Banweb interface, which is also used by the TCC
acctmngp program. Collisions are not common, but they will happen. For interface considerations, see
the internal document at /u/www/docs/tcc/projects/bannerfun/bannerfun.pdf: bannerfun:
Banner interface functions in Python.
#-- 1
try:
    self._imsUpload(MOOROSTER_CMD)
except CloseFailure as x:
    title = "mooroster failure"
    text = str(x)
    tkMessageBox.showerror(title, text)

14.19. App._saveOrCancel(): Insure changes have been saved

#-- 1
# [ if self._addsFile has unsaved changes ->
#    I
# else -> return POP_SAVE_SEMESTER ]
if not self._addsFile.isDirty():
    return POP_SAVE_SEMESTER

The popup is an instance of Section 27, “class MultiDialog: Dialog popup” (p. 173).

#-- 2
# [ reply := user choice from a popup with choices "Save"
#   or "Cancel" where "Cancel" returns POP_CANCEL ]
semesterName = class_sched.semesterName(
    self._addsFile.acadYear, self._addsFile.semCode)
text = POP_UNSAVED_CHOICES.format(semesterName)
saveText = POP_SAVE_SEMESTER.format(semesterName)
reply = MultiDialog(self, POP_UNSAVED_CHANGES, text,
    [saveText, POP_CANCEL])

#-- 3
if reply.result == POP_CANCEL:
    return POP_CANCEL
14.20. App._quitHandler(): The Quit button

The Quit button is not a command, just a request. Before we can quit, we must check two things:

- If there is a transient window in an invalid state, the user gets a popup asking them whether they want to discard those changes or cancel the request.
- If the current request set has unsaved changes, the user gets a popup asking them whether they want to save the file and quit, discard those changes and quit, or return to editing the current semester.

```python
# - - - A p p . _ q u i t H a n d l e r
def _quitHandler(self):
    '''Try to quit the application.'''
    #-- 1
    # [ if transient-can-close ->
    #   self := self with no transient displayed
    # else -> return ]
    try:
        self.closeTransient()
    except CloseFailure as x:
        return

For some unknown reason, self.quit() here does not quit the application, but falls through. Hence we resort to brute force: a SystemExit exception.

#-- 2
# [ if (self._addsFile is None) or
#   (self._addsFile has no unsaved changes) ->
#   stop execution
# else -> I ]
if ((self._addsFile is None) or (not self._addsFile.isDirty())):
    raise SystemExit
```

At this point we know there are unsaved changes. The user gets three choices.
#-- 3
# [ if the user chooses POP_DISCARD_SEMESTER in a popup ->
#   stop execution
# else if the user chooses POP_RETURN_TO_SEMESTER ->
#   return
# else ->
#   I ]
semesterName = class_sched.semesterName(self._addsFile.acadYear,
   self._addsFile.semCode)
saveFileName = addsFileName(self.semSched)
saveText = POP_SAVE_SEMESTER.format(  
   addFile(self.semSched))
returnText = POP_RETURN_TO_SEMESTER.format(semesterName)
reply = MultiDialog(self, POP_UNSAVED_CHANGES,
   POP_UNSAVED_CHOICES.format(semesterName),
   [saveText, POP_DISCARD_SEMESTER, returnText])
if reply.result == POP_DISCARD_SEMESTER:
   raise SystemExit
elif reply.result == returnText:
   return

Note
The “raise SystemExit” line above used to be “self.quit()”, but that method does not terminate
the application; it returns! The .quit() method is not overridden, and I have no idea why it returns
here.

At this point, the user has asked us to save their changes. This can still fail, for example if the save file
is unwriteable.

#-- 4
# [ if adds-file-name(self.semSched) can be written new ->
#   that file := self._addsFile as XML
#   self._addsFile := self._addsFile marked clean
#   stop execution
# else ->
#   (display) += error popup
#   return ]
try:
   self._saver()
   self.quit()
except CloseFailure:
   return

14.21. App._reqWidgets(): Create widgets in the primary instructor panel
This creates an instance of Section 18, “class Pager: Base widget with swappable content” (p. 116)
that holds the ReqPanel and makes it appear and disappear. For the constants that size the Pager,
see Section 9.31, “REQ_WIDE” (p. 30) and Section 9.32, “REQ_HIGH” (p. 30).
14.22. **App._swapWidgets()**: Create the swap area and its children

The swap area is an instance of Section 18, “**class Pager**: Base widget with swappable content” (p. 116) with the **SchedPanel**, **SectionPanel**, and **MetaPanel** as its transient children. For the size of this panel, see Section 9.33, “**SWAP_WIDE**” (p. 30) and Section 9.34, “**SWAP_HIGH**” (p. 30).

**Warning**

If the contents don’t fit in that size, they will be cropped. Be generous with these dimensions.
The `SectionPanel` class contains a second instance of the `UidFinder` panel. Because we don’t want this script to build the set of accounts in the TCC LDAP twice, there is a precondition here that `self._teacherFinder` has already been set up, so that we can pass the `acctList` keyword argument to the `UidFinder` constructor and it can reuse the account list from the first instance.

```python
#-- 3
# [ self._teacherFinder is a uidfinder.UidFinder instance ->
#   swapArea +=: a new SectionPanel whose role finder
#   widget reuses the account list from
#   self._teacherFinder
# self._sectionPanel := that SectionPanel ]
self._sectionPanel = SectionPanel(self, swapArea,
  finder=self._teacherFinder)
swapArea.add(self._sectionPanel)
```

Now create the `MetaPanel` transient. Whenever the user edits a metacourses’s title, it calls Section 14.45, “App.updateReqPanel(): Handler for a change in a metacourse name” (p. 88), which in turn calls Section 16.5, “ReqPanel.updateLists(): Update the section and metacourse lists” (p. 98), so that the list of metacourses displayed in the `ReqPanel` can be updated with the current metacourse title.

```python
#-- 4
# [ swapArea +=: a new MetaPanel instance
#   self._metaPanel := that MetaPanel instance ]
self._metaPanel = MetaPanel(swapArea, self,
  updater=self.updateReqPanel)
swapArea.add(self._metaPanel)

#-- 5
return swapArea
```

### 14.23. `App._addSectHandler()`: Add a section

The actual work of converting a `class_sched.SectionSchedule` instance to a `readadds.Section` instance is done by Section 16.2, “ReqPanel.addSection(): Add a section to the current instructor” (p. 96).

Note the precondition that `self._reqPanel` may not be `None`. The button that invokes this handler is on a `SchedPanel`, and a `SchedPanel` cannot be onscreen without having been invoked by the `ReqPanel`. Similarly, `self._semSched` is `None` only during initialization; we can’t get here until a semester has been selected.

```python
#-- 6
# [ (self._reqPanel is not None) and
#   (self._semSched is not None) and
#   (sectSched is a class_sched.SectionSchedule instance) ->
#   if there is a readadds.Section instance in self._addsFile
#   with the CRN from sectSched ->
#   (display) +=: a popup showing what teacher has
```
that request

```python
else ->
    self._reqPanel := self._reqPanel with a new
    readadds.Section instance added made from sectSched ]

#-- 1
    self._reqPanel.addSection(sectSched)
```

### 14.24. App.lookupDeptName(): Look up a department code

Another Law of Demeter pass-through.

```python
# - - - A p p . l o o k u p D e p t N a m e

def lookupDeptName(self, deptCode):
    '''Look up a department code.'''
    return self.classSchedule.lookupDeptName(deptCode)
```

### 14.25. App.lookupCRN(): Find a CRN in the current semester

```python
# - - - A p p . l o o k u p C R N

def lookupCRN(self, crn):
    '''Find the section with this CRN in the current semester.'''
    return self._addsFile.getCRN(crn)
```

### 14.26. App.lookupCRNSched(): Find a scheduled section by CRN

```python
# - - - A p p . l o o k u p C R N S c h e d

def lookupCRNSched(self, crn):
    '''Find a SectionSchedule in self.semSched by CRN.'''
    return self.semSched.lookupCrn(crn)
```

### 14.27. App.close(): Remove the current semester

```python
# - - - A p p . c l o s e

def close(self):
    '''Clear the ReqPanel and transient.'''
```

As discussed in Section 6.1, “Control flow” (p. 22), if a transient panel is showing and that panel is in an invalid state (for example, a metacourse that has not been named yet), we don’t want to remove the
transient panel unless we first alert the user in a popup and give them the choice to leave the transient panel open so they can fix the validity problem.

This method checks for that condition. If the condition described in specification function Section 10.12, "transient-can-close: Can the transient panel be removed?" (p. 41) is satisfied, it returns None, otherwise it raises a CloseFailure exception so that a requested operation (such as changing semesters) can be canceled.

This method also checks to see if self._addsFile, the AddsFile instance that holds the current set of course requests, has unsaved changes. If so, it uses a popup to give the user the choice of saving it to the adds-season-yyyy.xml file or discarding the changes. Note that this too can fail, if the save file is not writable; in that case it displays a popup and raises CloseFailure.

See Section 22.2, “SwapArea.close(): Blank the swap area” (p. 136).

```python
#-- 1
# [ if transient-can-close ->
#   self := self with no transient displayed
# else -> raise CloseFailure ]
self._swapArea.close()
```

See Section 16.4, “ReqPanel.close(): Remove this panel” (p. 98).

```python
#-- 2
# [ self := self with no ReqPanel displayed
# self._teacherFinder := self._teacherFinder cleared ]
self._reqPanel.close()
self._teacherFinder.clear()
```

Next we check for unsaved changes. If self._addsFile is None, no semester is currently displayed. Otherwise we check that AddsFile instance for unsaved changes; if there are none, set self._addsFile to None to indicate there is no current semester.

```python
#-- 3
if self._addsFile is None:
  return

#-- 4
# [ if self.addsFile has no unsaved changes ->
#   self.addsFile := None
#   return
# else -> I ]
if not self._addsFile.isDirty():
  self._addsFile = None
  return
```

At this point there is no transient panel, and there are unsaved changes in self._addsFile. Now we present a popup so the user can save the changes if they prefer: see Section 14.28, “App._saveOrDiscard(): Ask the user if they want to save their changes” (p. 77).

```python
#-- 5
# [ if the user selects "Save" in a popup ->
#   if adds-file-name(self.semSched) can
#   be written as new ->
```
Finally, set `self._addsFile` to `None` to signify that no semester is currently selected.

```python
self._addsFile = None
```

14.28. **App._saveOrDiscard()**: Ask the user if they want to save their changes

```python
def _saveOrDiscard(self):
    '''Ask the user to save changes or discard them.

    if the user selects "Save" in a popup ->
        if adds-file-name(self.semSched) can be written as new ->
            that file := self._addsFile as XML
        else ->
            (display) += an error popup
            raise CloseFailure
    else -> I ]

    #-- 1
    # [ if the user selects "Yes" in a popup ->
    #   I
    # else -> return ]
    semesterName = class_sched.semesterName(self.semSched.acadYear,
                                               self.semSched.semCode)
    fileName = addsFileName(self.semSched)
    saveText = POP_SAVE_SEMESTER.format(semesterName)
    reply = MultiDialog(self, POP_UNSAVED_CHANGES,
                         POP_UNSAVED_CHOICES.format(semesterName),
                         [saveText, POP_DISCARD_SEMESTER])
    if reply.result == POP_DISCARD_SEMESTER:
        return

    #-- 2
    # [ if adds-file-name(self.semSched) can be written new ->
    #   that file := self._addsFile as XML
    #   self._addsFile := self._addsFile marked clean
    # else ->
    #   (display) += error popup
    #   raise CloseFailure ]
    self._saver()```
14.29. **App.closeTransient: Clear the swap area**

This method is called by child panels to clear any transient panel from the swap area. It is at this level because the child panels don’t have direct control over the swap area.

```python
# - - -  A p p . c l o s e T r a n s i e n t
def closeTransient(self):
    '''Clear the swap area, unless the user cancels.
    '''
    #-- 1
    # [ if transient-can-close ->
    #   self := self with no transient displayed
    # else -> raise CloseFailure ]
    self._swapArea.close()
```

14.30. **App.openSemester(): Switch to a different semester**

This method reads the `adds-season-yyyy.xml` file for a different semester, if there is one, or creates a new `AddsFile` instance if the user is just starting to accumulate requests for a new semester. The argument is a `class_sched.SemesterSchedule` instance. See Section 28, “addsFileName(): The name of the save file” (p. 175) for the logic that determines the `adds-season-yyyy.xml` file name.

```python
# - - -  A p p . o p e n S e m e s t e r
def openSemester(self, newSched):
    '''Close the current semester, open a new semester.'''
    #-- 1
    # [ newAddsFileName := adds-file-name(newSched) ]
    newAddsFileName = addsFileName(newSched)

    if not os.path.exists(newAddsFileName):
        title = "Starting a new semester"
        text = ("This directory '{0}' has no '{1} file: "
                "creating a new, empty semester.").format(
                os.getcwd(), newAddsFileName)
```

If the file doesn’t exist, we show an informational popup informing the user that they are starting up a new, empty semester, and set `self._addsFile` to a new, empty `AddFile` instance. See also Section 14.31, “App._showSemester(): Change semester-dependent widgets” (p. 79), which changes semester-related widgets.
At this point we know that there is an `adds-season-yyyy.xml` file, but we might not be able to read it, or it might not be valid against the schema.

---

14.31. `App._showSemester()`: Change semester-dependent widgets

The purpose of this method is to change the two widgets that display the current semester name: `self._semPicker`, the semester picker widget, and `self._saveTitle`, the `StringVar` that controls what appears on the `Save adds-season-yyyy.xml` button.
newFileName = addsFileName(semSched)
self._saveTitle.set("Save {0}".format(newFileName))

14.32. App._saver(): Write back the XML

This internal method is used by various other controls to save the file to the appropriate name for the current semester. It also saves an archive copy, building the archive subdirectory if necessary.

```python
#-- App._saver

def _saver(self):
    '''Write back the current requests set as XML.

    [ self.semSched is not None ->
        if (adds-file-name(self.semSched.acadYear,
            self.semSched.semCode) can be written new) and
        and (arch-file(self.semSched.acadYear,
            self.semSched.semCode) can be written new) ->
            those files := self.addsFile as XML
            self.addsFile := self._addsFile marked clean
        else ->
            (display) +=: error popup
            raise CloseFailure ]
    ..
    
See Section 28, "addsFileName(): The name of the save file" (p. 175) and Section 14.34, "App._archFileName(): Generate a name for the archive file" (p. 82).
```

```python
#-- 1
# [ outFileName := adds-file-name(acadYear, semCode)
# archFileName := arch-file(acadYear, semCode) ]
outFileName = addsFileName(self.semSched)
archFileName = self._archFileName(self.semSched)

We write the archive file before we write the adds-season-yyyy.xml file. First we create the subdirectory if it doesn't exist. If it exists but isn't a subdirectory or isn't writeable, that will become apparent later when we try to save the archive file.

```python
#-- 2
# [ if the directory containing archFileName does not exist ->
#   if that directory can be created ->
#     (file system) +=: that directory
#   else ->
#     (display) +=: error popup
#     raise CloseFailure
#   else -> I ]
try:
    self._makeArchDir(archFileName)
except CloseFailure as x:
    title = "Can't create archive subdirectory"
    text = ("Unable to create subdirectory '{0}': "
        "{1}".format(archDirName, str(x)))
```

```
Now try to save to the archive file.

```python
try:
    self._addsFile.save(archFileName)
except IOError as x:
    title = "Can't create archive adds-file"
    text = ("Unable to create archive file '{0}': {1}".format(archFileName, str(x)))
    tkMessageBox.showerror(title, text)
    raise CloseFailure(str(x))
```

Finally, save the working adds-season-yyyy.xml file.

```python
try:
    self._addsFile.save(outFileName)
except IOError as x:
    title = "Can't save your changes"
    text = ("Unable to save '{0}': {1}".format(outFileName, str(x)))
    tkMessageBox.showerror(title, text)
    raise CloseFailure(str(x))
```

### 14.33. App._archDirName(): Archive directory name

Archive copies of adds-season-yyyy.xml and IMS Enterprise sects_.files are grouped by semester in subdirectories of the current working directory, for example, "arch-fall-2017". See Section 10.2, “arch-dir-name: Archive directory name” (p. 39).

```python
# - - - A p p . _ a r c h D i r N a m e

def _archDirName(self, acadYear, semCode):
    """Name of a given semester's archive subdirectory.

    [ (acadYear is a four-digit academic year as a str) and
      (semCode is a semester code) ->
      return arch-dir-name(acadYear, semCode) ]
    ...
```
14.34. **App._archFileName(): Generate a name for the archive file**

Implements the intended function in Section 10.3, “arch-file: Name of an archival backup file” (p. 39).

```python
def _archFileName(self, semSched):
    """Generate the name for an archived copy of the adds file.

    [ semSched is a class_sched.SemesterSchedule ->
      return arch-file(semSched.acadYear, semSched.semCode) ]

    """
    #-- 1
    acadYear = semSched.acadYear
    semCode = semSched.semCode

    #-- 2
    # [ calYear := cal-year(acadYear, semCode)
    #    semName := semester-name(self.semSched.semCode).lower()
    calYear = class_sched.acadYearToCal(acadYear, semCode)
    semName = class_sched.SEM_CODE_TO_NAME[semCode].lower()

    #-- 3
    # [ archFileName := archDirName + ADDS_PREFIX + semName +
    #    calYear + stamp + XML_EXT ]
    archFileName = ("{0}{1}{2}-{3}-{4}{5}".format(
        archDirName, ADDS_PREFIX, semName, calYear,
        stamp, XML_EXT))

    #-- 5
    return archFileName
```

14.35. **App._makeArchDir(): Build the archive directory if necessary**
def _makeArchDir(self, archFileName):
    '''Create the archive directory if necessary.

    [ archFileName is a pathname ->
      if the directory containing archFileName exists -> I
      else if that directory can be created ->
        (file system) += that directory
      else -> raise CloseFailure ]
    ...
    #-- 1
    # [ archDirName := the portion of archFileName up to but
    #   not including the last '/' ]
    archDirName, discard = os.path.split(archFileName)

    #-- 2
    # [ if archDirName exists ->
    #   return
    #   else -> I ]
    if os.path.exists(archDirName):
        return

    #-- 3
    # [ if the user can create directory (archDirName) ->
    #   (file system) += that directory
    #   else -> raise CloseFailure ]
    try:
        os.mkdir(archDirName)
    except (OSError, IOError) as x:
        raise CloseFailure("Can't create the archive "
                            "subdirectory '{0}': {1}".format(archDirName, str(x)))

14.36. App.reqBlank(): Blank out the ReqPanel

def reqBlank(self):
    '''Blank self._reqPager.''

    self._reqPager.clear()

14.37. App.reqSelect(): Display a selected instructor

def reqSelect(self, teacher):
    '''Display the given teacher in the ReqPanel.''

    self._reqPager.select(self._reqPanel)
    self._reqPanel.select(teacher)
14.38. **App.removeInstructor()**: Delete a primary instructor's records

If the instructor has any `readadds.Course` or `readadds.CrossParent` children, we'll need to ask the user whether it's okay to delete.

```python
# - - - A p p . r e m o v e I n s t r u c t o r

def removeInstructor(self, uid):
    '''Remove the instructor from self._addsFile.
    '''
    try:
        teacher = self._addsFile.getTeacher(uid)
        return
    except KeyError:
        return
```

Since the button that invokes this function is on the `ReqPanel`, there should be a corresponding `readadds.Teacher` instance, so the lookup failure in this next block shouldn't happen.

```python
#-- 1
# [ if self._addsFile has a readadds.Teacher instance with
#   email=(uid) ->
#   teacher := that instance
#   else -> return ]
try:
    teacher = self._addsFile.getTeacher(uid)
except KeyError:
    return
```

Now we can proceed to remove the `Teacher` instance from `self._addsFile` and close the panel. First we close the transient panel if there is one. The transient panel might not ordinarily be closeable, but since the transient panel is under control of the `ReqPanel`, and since this `ReqPanel` is going away shortly, we can use the low-level method to close the transient without worrying about whether its `.close()` method would fail due to invalid data. The `.clear()` method here is actually Section 18.3, “Pager.clear(): Erase the window” (p. 118).

```python
#-- 2
# [ self := self with no transient displayed ]
try:
    self.closeTransient()
except CloseFailure:
    self._swapArea.clear()
#-- 5
# [ self._reqPager := self._reqPager cleared ]
self.reqBlank()
#-- 6
# [ self._addsFile := self._addsFile with (teacher) removed ]
self._addsFile.delTeacher(teacher.email)
```

14.39. **App.showDept()**: Display the schedule for one department

This operation can fail:

- What we get as an argument is a department code. There is no guarantee that the current semester's schedule (`self.semSched`) has any sections in that department.
Because this method needs to use the transient area, it can fail (see Section 6.1, “Control flow” (p. 22).
Assuming we can use the transient area, tell the swap area to display the SchedPanel, and then tell
that panel to display the given class_sched.DeptSched.

```python
# - - - A p p . s h o w D e p t

def showDept(self, deptCode):
    '''Display a class_sched.DeptSchedule in the swap area.
    '''
    #-- 1
    # [ if self.semSched contains any sections for department
    #   code (deptCode) ->
    #   deptSched := the class_sched.DeptSchedule instance
    #     containing those sections
    # else ->
    #   (display) := an error popup
    #   return ]
    try:
        deptSched = self.semSched.lookupDept(deptCode)
    except KeyError:
        title = "No courses scheduled"
        text = ("There are no courses scheduled for the
        \{dept\} department during \{sem\}.”.format( 
            dept=deptCode, 
            sem=self.semSched.semName))
        tkMessageBox.showerror(title, text)
        return
    #-- 2
    # [ if transient-can-close ->
    #   self := self displaying no transient
    # else -> raise CloseFailure ]
    self.closeTransient()
    #-- 3
    # [ self._swapArea := self._swapArea displaying
    #   self._schedPanel ]
    self._swapArea.select(self._schedPanel)
    #-- 4
    # [ self._schedPanel := self._schedPanel displaying
    #   department (deptSched) for semester (self.semSched) ]
    self._schedPanel.select(self.semSched, deptSched)
```

14.40. App.showSection(): Bring up the SectionPanel

This method attempts to display the SectionPanel transient and passes a readadds.Section instance
to display in it.
def showSection(self, section):
    '''Select the SectionPanel transient.
    ...'
    #-- 1
    # [ if transient-can-close ->
    #   self := self with no transient displayed
    # else -> return ]
    try:
        self.closeTransient()
    except CloseFailure:
        return

See Section 18.2, “Pager.select(): Make a child appear” (p. 118), which SwapArea inherits; see also Section 25.16, “SectionPanel.select(): Display a readadds.Section instance” (p. 158).

def showMeta(self, crossParent):
    '''Bring up the MetaPanel editing a CrossParent
    ...'
    #-- 1
    # [ if transient-can-close ->
    #   self := self with no transient displayed
    # else -> return ]
    try:
        self.closeTransient()
    except CloseFailure:
        return

See Section 18.2, “Pager.select(): Make a child appear” (p. 118), which SwapArea inherits; see also Section 26.1, “MetaPanel.select(): Display a metacourse” (p. 164).
14.42. App.deleteMeta(): Remove a metacourse

This method is used to remove an invalid metacourse from the current ReqPanel and also to blank the swap area.

```python
# -- - - - A p p . d e l e t e M e t a

def deleteMeta(self, crossParent):
    '''Discard a metacourse.'''
    #-- 1
    # [ (self._reqPanel is a ReqPanel instance) and
    # (crossParent is a readadds.CrossParent in self._reqPanel) ->
    # self._reqPanel := self._reqPanel with crossParent
    # removed from its display and from
    # self._reqPanel.teacher ]
    self._reqPanel.deleteMeta(crossParent)
    #-- 2
    # [ self := self displaying no transient ]
    self._swapArea.clear()
```

14.43. App.getSelectedSection(): What section is selected in the ReqPanel?

Used by the Add selected section on the MetaPanel to find out what section is highlighted in the ReqPanel.

```python
# -- - - - A p p . g e t S e l e c t e d S e c t i o n

def getSelectedSection(self):
    '''What section is selected in the instructor panel?'''
    #-- 1
    # [ if the sections list in self._reqPanel has a line
    # selected ->
    # return the readadds.Section for that line
    # else -> raise ValueError ]
    return self._reqPanel.getSelSection()
```

14.44. App.clearSpecificMeta()

This method is called when a metacourse is about to be destroyed. It insures that, if the swap area is currently displaying that metacourse, it is forcibly cleared, whether or not it is in a valid state. See Section 18.4, “Pager.get(): Return the currently displayed widget” (p. 119); Section 26.12, “MetaPanel.clear(): Clear out the content” (p. 171); and Section 18.3, “Pager.clear(): Erase the window” (p. 118) (which SwapArea inherits).
def clearSpecificMeta(self, crossParent):
    '''If the SwapArea is showing crossParent, clear it.'''
    #-- 1
    # [ if self._swapArea is not displaying self._metaParent ->
    #     return
    # else -> I ]
    current = self._swapArea.get()
    if current != self._metaPanel:
        return

    #-- 2
    # [ self._metaPanel := self._metaPanel, cleared ]
    self._metaPanel.clear()

    #-- 3
    # [ self._swapArea := self._swapArea not displaying a
    #     transient ]
    self._swapArea.clear()

14.45. App.updateReqPanel(): Handler for a change in a metacourse name

This method is called when the user changes the name of a section in the SectionPanel or changes the name of a metacourse in the MetaPanel. Its purpose is to inform the ReqPanel of that change so that it can update its lists of sections and metacourses. See Section 16.5, “ReqPanel.updateLists(): Update the section and metacourse lists” (p. 98).

def updateReqPanel(self):
    '''Request ReqPanel to rebuild its section/metacourse lists.'''
    #-- 1
    # [ if self._reqPager is displaying self._reqPanel ->
    #     self._reqPanel := self._reqPanel with its lists of
    #     sections and metacourses rebuilt from its
    #     current teacher ]
    if self._reqPager.get() is self._reqPanel:
        self._reqPanel.updateLists()
command is the external semester name, the `SemesterSchedule.semName` attribute such as “Spring 2020”.

This widget may attempt to close the transient panel, which can fail; refer to the specification function Section 10.10, “semester-picker-callback: Callback for change of semester” (p. 41) for a discussion of the related issues.

```python
# - - - - - c l a s s  s e m e s t e r  p i c k e r

class SemesterPicker(ttk.LabelFrame):
    '''Semester selector compound widget.

    Exports:
    SemesterPicker(parent, classSched, callback=None):
        [ (parent is a ttk.Frame) and
          (classSched is a class_sched.ClassSchedule instance) and
          (callback is a semester-picker-callback) ->
          parent += a new SemesterPicker widget, unregistered,
          displaying the semesters in classSched and calling
          (callback) when the user requests a new semester
          return that new SemesterPicker widget ]
    .classSched: [ as passed to constructor, read-only ]
    .callback:  [ as passed to constructor, read-only ]
    .semSched:  [ currently selected SemesterSchedule ]
    .set(semSched):
        [ semSched is a SemesterSchedule in self.classSched ->
          if (self.callback is not None) and
          (self.callback(semSched) raises CloseFailure ->
            raise CloseFailure
          else ->
            self := self displaying semSched ]

    State/Invariants:
    ._acadYear:   [ self's current academic year or None ]
    ._semCode:    [ self's current semester code or None ]

    Widgets:
    ._mb:        [ a ttk.Menubutton ]
    ._mbVar:
        [ a tk.StringVar for the title of self._mb ]
    ._menu:      [ the tk.Menu for self._mb ]

15.1. SemesterPicker.set(): Change the displayed semester

This method changes the currently displayed semester name on the ttk.Menubutton.

# - - -  s e m e s t e r  p i c k e r . s e t

def set(self, semSched):
    '''Display a different semester.
```
15.2. SemesterPicker.__init__()  

```python
# - - - S e m e s t e r P i c k e r . __ i n i t _ _

def __init__(self, parent, classSched, callback=None):
    '''Constructor.
    '''
    #-- 1
    # [ self := self as a ttk.LabelFrame ]
    ttk.LabelFrame.__init__(self, parent,
                            text=CHANGE_SEMESTER_FRAME)
    self.grid()
    #-- 2
    self.classSched = classSched
    self.callback = callback
    self._acadYear = None
    self._semCode = None
    #-- 3
    # [ self := self with all widgets and variables created ]
    self._createWidgets()
```

15.3. SemesterPicker._createWidgets()  

```python
# - - - S e m e s t e r P i c k e r . _ c r e a t e W i d g e t s

def _createWidgets(self):
    '''Create all widgets and associated control variables.
    '''
    #-- 1
    # [ self._mbVar := a new tk.StringVar
    # self := self with a new ttk.Menubutton added and
    # registered and a new tk.Menu connected to it
    # self._mb := that ttk.Menubutton
    # self._menu := that tk.Menu ]
    self._mbVar = tk.StringVar()
    self._mb = ttk.Menubutton(self,
                              textvariable=self._mbVar,
                              width=len("Summer 9999"))
    self._mb.grid(padx=5, pady=5)
    self._menu = tk.Menu(self._mb, tearoff=0)
    self._mb['menu'] = self._menu
    #-- 2
    # [ self._menu += commands corresponding to the semesters
```
# in self.classSched, each of which calls self.callback (if defined) and changes self to display that semester, unless the callback raises CloseFailure ]
for semSched in self.classSched.genSemesters():
    self._addSemester(semSched)

15.4. SemesterPicker._addSemester(): Add one semester command to the semester menu

def _addSemester(self, semSched):
    '''Add a choice to the semester menu.

    [ semSched is a class_sched.SemSchedule in self.classSched ->
    self._menu +=: a menu command that displays semSched.semName and, when clicked, calls self.callback (if defined) and changes self to display semester (semSched), unless the callback raises CloseFailure ]
    '''

    This method uses tk.Menu.add_command() to add a new choice to self._menu. We define a closure here as the command handler. This closure must handle a possible CloseFailure exception; see the specification function Section 10.10, “semester-picker-callback: Callback for change of semester” (p. 41) for a discussion of the protocol for this callback.

    #-- 1
    # [ handler := a function that calls self.callback if defined, and returns if that callback raises CloseFailure; if self.callback returns normally, or if there is no callback, it sets self to display semester (semSched) ]
    def handler():
        '''Handler for change of semester.
        '''

        #-- 1
        # [ if (self.callback is not None) and (self.callback(semSched) raises CloseFailure) ->
        #   return
        #   else -> I ]
        if self.callback is not None:
            try:
                self.callback(semSched)
            except CloseFailure:
                return

        #-- 2
        # [ self := self displaying semester (semSched) ]
        self.set(semSched)
That’s the end of the closure. Now add the new menu command using that as the command handler.

```python
#-- 2
# [ self._menu += a new command displaying semSched.semName
#     and calling handler ]
self._menu.add_command(command=handler, label=semSched.semName)
```

16. class ReqPanel: The primary instructor edit panel

This widget grouping appears whenever the user selects a primary instructor in the *Click to show an instructor’s courses* UidFinder. It disappears when the current semester is closed: for example, when the user changes semesters, this widget will not appear until the user selects an instructor. The class is named ReqPanel because it operates on the internal equivalent of the `req` element in the `adds-season-yyyy.xml` file.

```python
class ReqPanel(ttk.Frame):
    '''Primary instructor (req element) edit panel.

    Exports:
    ReqPanel(pager, app, **kw):
        [ (parent is a Pager instance) and
        (app is the containing App instance) ->
        pager := pager with a new ReqPanel widget added,
        but not displayed ]
    .pager: [ as passed to constructor, read-only ]
    .app: [ as passed to constructor, read-only ]
    .teacher:
        [ if no teacher has been selected -> None
        else -> current readadds.Teacher instance ]
    .select(teacher):
        [ teacher is a readadds.Teacher instance ->
        if (transient-can-close) and
        (self is in a valid state) ->
        app := app with no transient showing
        self := self directly editing teacher
        else -> raise CloseFailure ]
    .addSection(sectSched):
        [ (self.teacher is a readadds.Teacher instance) and
        (sectSched is a class_sched.SectionSchedule instance) ->
        if there is an existing request for CRN (sectSched.crn)
        in self.app ->
        (display) +=: a popup showing what teacher has
        that request
        else ->
        self.teacher := self.teacher with a new
        readadds.Section instance added, including its
        parent readadds.Course instance if necessary ]
    .getSelSection():
```
[ if self's section list has a section selected -> 
  return the corresponding readadds.Section instance 
 else -> raise ValueError ]

.close():
  [ if (transient-can-close) and 
    (self is in a valid state) ->
    self.app := self.app not showing a transient or self
  else -> raise CloseFailure ]

.updateLists():
  [ self := self with its lists of sections and metacourses 
    updated from self.teacher ]

deleteMeta(crossParent):
  [ crossParent is a readadds.Crossparent in self.teacher ->
    self.teacher := self.teacher without crossParent
    self := self with its lists of sections and 
    metacourses updated from (self.teacher without 
    crossParent) ]

In this enumeration of the Tkinter-related attributes, the indentation shows which widgets are inside which.

Widgets/Control variables:
  .mainLabel:
    [ a ttk.Label identifying this panel ]
  .mainLabelVar:
    [ the StringVar textvariable for ._mainLabel ]
  .editNameFrame:
    [ a LabelFrame enclosing the widgets for editing the 
      instructor's name ]
  .givenLabel:
    [ ttk.Label for self._givenEntry ]
  .givenEntry:
    [ ttk.Entry to edit teacher.given ]
  .givenVar:
    [ tk.StringVar for self._givenEntry ]
  .middleLabel:
    [ ttk.Label for self._middleEntry ]
  .middleEntry:
    [ ttk.Entry to edit teacher.middle ]
  .middleVar:
    [ tk.StringVar for self._middleEntry ]
  .familyLabel:
    [ ttk.Label for self._familyEntry ]
  .familyEntry:
    [ ttk.Entry to edit teacher.family ]
  .familyVar:
    [ tk.StringVar for self._familyEntry ]
  .controlFrame:
    [ ttk.Frame for the following buttons ]
  .removeInstrButton:
    [ ttk.Button to remove this instructor from the semester ]
  .closeButton:
    [ ttk.Button to close and unregister self ]
Grid plan for self:
+-----------------+
| ._mainLabel     |
+-----------------+
| ._editNameFrame |
+-----------------+
| ._controlFrame  |
+-----------------+
| ._sectionsFrame |
+-----------------+
| ._metaFrame     |
+-----------------+

Grid plan for ._editNameFrame:
+-----------------------------+
0 | ._givenLabel | ._givenEntry |
+-----------------------------+
1 | ._middleLabel | ._middleEntry |
+-----------------------------+
2 | ._familyLabel | ._familyEntry |
+-----------------------------+

Grid plan for ._controlFrame:
+-----------------------------+
0 | ._removeInstrButton | ._closeButton |
+-----------------------------+

Grid plan for ._sectionsFrame:
Here are some class constants. \texttt{SPACER \_WIDTH} controls the extra space between the label/entry pairs for \texttt{self._editNameFrame}; the next three give the sizes of the edit boxes for names.

\begin{verbatim}
SPACER WIDTH = '10'
GIVEN WIDTH = '15'
MIDDLE WIDTH = '15'
FAMILY WIDTH = '20'
\end{verbatim}

This constant defines the font size for the entry fields.

\begin{verbatim}
ENTRY_FONT_SIZE = 14
\end{verbatim}

### 16.1. \texttt{ReqPanel.select()}: Display a given instructor

Given an instructor’s set of course requests as a \texttt{readads.Teacher} instance, this method displays that instance. It can fail; see Section 16.4, “\texttt{ReqPanel.close()}: Remove this panel” (p. 98) for possible causes.

\begin{verbatim}
#-- - ReqPanel.select

def select(self, teacher):
    '''Display one instructor's set of course requests."
    ...
    #-- 1
    # [ if (transient-can-close) and
    # (self is in a valid state) ->
    # self.app := self.app not showing a transient or self
    # else -> raise CloseFailure ]
    self.close()

Now that the panel has been cleared (and has disappeared from the display), populate the fields that are attributes of the instructor. First, modify the panel’s (read-only) main label to show the instructor’s email and full name. Then distribute the three pieces of the name to their appropriate fields.

\begin{verbatim}
#-- 2
# [ self.teacher := teacher
\end{verbatim}
Now make the panel appear; see Section 18.2, “Pager.select(): Make a child appear” (p. 118).

Finally, reload the section and metacourse listboxes; see Section 16.5, “ReqPanel.updateLists(): Update the section and metacourse lists” (p. 98).

16.2. ReqPanel.addSection(): Add a section to the current instructor

The course code is the department code plus one space plus the course number; the section code is the section type plus the section number.

```python
#-- 3
# [ self.pager := self.pager displaying self ]
self.pager.select(self)
```

```python
#-- 4
# [ self := self with its lists of sections and metacourses
#   rebuild from self.teacher ]
self.updateLists()
```

```python
#-- ------- ReqPanel.addSection

def addSection(self, sectSched):
    """Add a new readadds.Section.
    """
    #-- 1
    # [ if there is an existing request for CRN (sectSched.crn)
    #   in self.app ->
    #   (display) +=: a popup showing what teacher has that
    #   request
    #   return
    # else -> I ]
    try:
        section = self.app.lookupCRN(sectSched.crn)
        teacher = section.teacher()
        title = "Duplicate section request"
        text = ("CRN {sect.crn} ({sect.fullCode}, {sect.desc})"
                "is already assigned to "
                "{teach.fullName} ({teach.email})".format(
                    sect=section, 
                    teach=teacher))
        tkMessageBox.showerror(title, text)
```
return
eexcept KeyError:
    pass

#-- 2
courseCode = "{0} {1}".format(sectSched.deptCode,
    sectSched.courseNo))
sectionCode = sectSched.sectType + sectSched.sectNo

#-- 2
#[ if self.teacher has a course with code (courseCode) ->
#  course := the corresponding readadds.Course instance
# else ->
#  self.teacher += a new readadds.Course instance
#  with code (courseCode) and an empty description
#  course := that new readadds.Course instance ]
try:
    course = self.teacher.getCourse(courseCode)
except KeyError:
    course = readadds.Course(self.teacher, courseCode)

#-- 3
#[ course += a new readadds.Section instance with CRN
#  sectSched.crn, code (sectionCode), desc (sectSched.title),
#  and no reload information ]
newSect = readadds.Section(course, sectSched.crn, sectionCode,
    sectSched.title, None)

#-- 4
#[ self._sectionsList += newSect ]
self._sectionsList.add(newSect)

16.3. ReqPanel.getSelSection(): Which section is selected?

def getSelSection(self):
    '''Return the selected readadds.Section.
    '''
    #-- 1
    section = self._sectionsList.get()
    if section is None:
        raise ValueError("No section is selected in the "
            "instructor's list of section requests.")
    else:
        return section
16.4. `ReqPanel.close()`: Remove this panel

This method is the handler for the Close button, but it may also be called at other times, such as when the user is changing to a different semester.

The close operation removes the panel from the display. However, it can fail: if there is a transient panel showing that is in an invalid state; or if there is not at least a family name in the Family entry.

This class does not have any direct knowledge of the transient panel; it delegates the removal operation to Section 14.29, “App.closeTransient: Clear the swap area” (p. 78).

```python
def close(self):
    '''Try to remove self (and any transient) from the display.
    --- 1
    if self.teacher is None:
        return  # Already closed
    --- 2
    if (app has no transient) or
    (app's transient is in a valid state) or
    (the user elects to discard those changes via popup) ->
    app := app with no transient displayed
    else -> raise CloseFailure
    self.app.closeTransient()
    --- 3
    if self._familyVar is empty ->
    raise CloseFailure
    else ->
    self.pager := self.pager cleared
    self := self, unregistered, with self.teacher=None,
    and empty section and metacourse lists

    if self._familyVar.get().strip() == '':
        raise CloseFailure("Please enter the instructor's "
            "given, middle, and family name.")
    else:
        self.pager.clear()
        self.teacher = None
        self._sectionsList.clear()
        self._metaList.clear()
```

16.5. `ReqPanel.updateLists()`: Update the section and metacourse lists

This method is called when the user changes the name of a section in the SectionPanel transient, or when they change the name of a metacourse in the MetaPanel transient. The method’s purpose is to rebuild the listboxes displaying the names of the instructor’s sections and metacourses.

```python
def updateLists(self):
```

---

98  cmsadds: A GUI for CMS course creation  New Mexico Tech Computer Center
''Update self's section and metacourse lists.
''

```python
#-- 1
# [ self._sectionsList := self._sectionsList cleared
#   self._metaList := self._metaList cleared ]
self._sectionsList.clear()
self._metaList.clear()

#-- 2
# [ self._sectionsList += sections in self.teacher ]
for course in self.teacher.genCourses():
    for section in course.genSections():
        self._sectionsList.add(section)

#-- 3
# [ self._metaList += metacourses in self.teacher ]
for cross in self.teacher.genCrosses():
    self._metaList.add(cross)
```

### 16.6. ReqPanel.deleteMeta(): Remove a metacourse

```python
# - - - R e q P a n e l . d e l e t e M e t a
def deleteMeta(self, crossParent):
    '''Remove a metacourse.
    '''
    #-- 1
    # [ self.teacher := self.teacher without crossParent ]
    self.teacher.remove(crossParent)

    #-- 2
    # [ self := self with its lists updated from self.teacher ]
    self.updateLists()
```

### 16.7. ReqPanel.__init__()
self.app = app
self.teacher = None

See Section 16.8, “ReqPanel._createWidgets()” (p. 100).

```python
#-- 2
# [ self := self with all widgets and control variables ]
self._createWidgets()
```

**16.8. ReqPanel._createWidgets()**

```python
#---- ReqPanel._createWidgets

def _createWidgets(self):
    '''Create all self's widgets.
    '''

Refer to the grid plan in Section 16, “class ReqPanel: The primary instructor edit panel” (p. 92).

```python
#-- 2
# [ self := self with a new ttk.Label added and registered
#    self.mainLabelVar := a new tk.StringVar initially
#    set to INSTR_LABEL
#    self._mainLabel := that ttk.Label ]
rowx = 0
self.mainLabelVar = tk.StringVar()
self.mainLabelVar.set(INSTR_LABEL)
self._mainLabel = ttk.Label(self, text=INSTR_LABEL,
                         style=PANEL_TITLE_STYLE,
                         textvariable=self.mainLabelVar)
self._mainLabel.grid(row=rowx, sticky=tk.W)

See Section 16.9, “ReqPanel._createEditNameFrame(): Create the name-editing widgets” (p. 101).

```python
#-- 3
# [ self := self with a new ttk.Frame added and registered
#    containing the name-editing widgets
#    self.editNameFrame := that ttk.Frame ]
rowx += 1
self.editNameFrame = self._createEditNameFrame()
self.editNameFrame.grid(row=rowx, sticky=tk.W)

See Section 16.11, “ReqPanel._createControlFrame(): Add panel controls” (p. 103).

```python
#-- 4
# [ self := self with a new ttk.Frame added and registered
#    containing the panel controls
#    self.controlFrame := that ttk.Frame ]
rowx += 1
self.controlFrame = self._createControlFrame()
self.controlFrame.grid(row=rowx, sticky=tk.W, pady=5)
```
See Section 16.13, “ReqPanel._createSectionsFrame(): Create the section list area” (p. 106).

```python
#-- 5
#: [ self := self with a new ttk.Frame added and registered
#: containing the section-related widgets ]
rowx += 1
self._sectionsFrame = self._createSectionsFrame()
self._sectionsFrame.grid(row=rowx, sticky=tk.W,
                          padx=10, pady=10)
```

See Section 16.18, “ReqPanel._createMetaFrame(): Create the metacourse widget group” (p. 111).

```python
#-- 6
rowx += 1
self._metaFrame = self._createMetaFrame()
self._metaFrame.grid(row=rowx, sticky=tk.W,
                      padx=10, pady=10)
```

16.9. ReqPanel._createEditNameFrame(): Create the name-editing widgets

```python
def _createEditNameFrame(self):
    '''Create the name-editing widgets.
    '''
    #-- 1
    #: [ self := self with a new ttk.Frame added ]
    #: f := that ttk.Frame ]
    f = ttk.Frame(self)

    #: [ f := f with a new ttk.Label added showing GIVEN_LABEL
    #: self._givenLabel := that ttk.Label ]
    rowx, colx = 0, 0
    self._givenLabel = ttk.Label(f, text=GIVEN_LABEL)
    self._givenLabel.grid(row=rowx, column=colx, sticky=tk.E)
    
    #: [ self._givenVar := a new tk.StringVar
    #: f := f with a ttk.Entry added using that tk.StringVar
    #-- 3
    #: [ self._givenVar := a new tk.StringVar
    #: f := f with a ttk.Entry added using that tk.StringVar
```

When the user makes changes to any of the three name fields, we’ll need to save those changes to the underlying readadds.Teacher instance. This is done by binding the FocusOut event on those Entry widgets to the handler Section 16.10, “ReqPanel._updateNames(): Save the contents of the name fields” (p. 103). That event occurs whenever focus leaves a widget. This means that the handler will be called even if the user tabs through a field without changing it, but that is okay because the value they see will be the value saved.

Rather than creating three separate event handlers, all three widgets call the same handler, and the handler saves the current values of all three fields.

```python
#: [ self._givenVar := a new tk.StringVar
#: f := f with a ttk.Entry added using that tk.StringVar
```

New Mexico Tech Computer Center  cmsadds: A GUI for CMS course creation  101
# that saves self._givenVar, self._middleVar, and # self._familyVar to self.teacher on FocusOut 
# self._givenEntry := that ttk.Entry ]
colx += 1
self._givenVar = tk.StringVar()
self._givenEntry = ttk.Entry(f, width=self.GIVEN_WIDTH,
font=self._entryFont,
textvariable=self._givenVar)
self._givenEntry.grid(row=rowx, column=colx,
sticky=tk.W, padx=5)
self._givenEntry.bind("<FocusOut>", self._updateNames)

#-- 4
# [ f := f with a new ttk.Label added showing MIDDLE_LABEL
# self._middleLabel := that ttk.Label ]
rowx, colx = rowx+1, 0
self._middleLabel = ttk.Label(f, text=MIDDLE_LABEL)
self._middleLabel.grid(row=rowx, column=colx, sticky=tk.E)

#-- 5
# [ self._middleVar := a new tk.StringVar
# f := f with a ttk.Entry added using that tk.StringVar
# that saves self._givenVar, self._middleVar, and # self._familyVar to self.teacher on FocusOut
# self._middleEntry := that ttk.Entry ]
colx += 1
self._middleVar = tk.StringVar()
self._middleEntry = ttk.Entry(f, width=self.MIDDLE_WIDTH,
font=self._entryFont,
textvariable=self._middleVar)
self._middleEntry.grid(row=rowx, column=colx,
sticky=tk.W, padx=5)
self._middleEntry.bind("<FocusOut>", self._updateNames)

#-- 6
# [ f := f with a new ttk.Label added showing FAMILY_LABEL
# self._familyLabel := that ttk.Label ]
rowx, colx = rowx+1, 0
self._familyLabel = ttk.Label(f, text=FAMILY_LABEL)
self._familyLabel.grid(row=rowx, column=colx, sticky=tk.E)

#-- 7
# [ self._familyVar := a new tk.StringVar
# f := f with a ttk.Entry added using that tk.StringVar
# that saves self._givenVar, self._middleVar, and # self._familyVar to self.teacher on FocusOut
# self._familyEntry := that ttk.Entry ]
colx += 1
self._familyVar = tk.StringVar()
self._familyEntry = ttk.Entry(f, width=self.FAMILY_WIDTH,
font=self._entryFont,
textvariable=self._familyVar)
self._familyEntry.grid(row=rowx, column=colx,
16.10. **ReqPanel._updateNames(): Save the contents of the name fields**

This event handler is called whenever there is a FocusOut event on one of the three name fields. Its purpose is to save the contents of those three fields to `self.teacher`.

There must be a value in at least the family name field. If there isn’t, display a popup and move the focus back into the family name field until the user fixes it.

```python
def _updateNames(self, event):
    '''Handle a FocusOut on one of the name Entry fields.

    [ event is a tk.Event instance ->
      self.teacher.given := value of self._givenVar
      self.teacher.middle := value of self._middleVar
      self.teacher.family := value of self._familyVar ]
    ...
    #-- 1
    family = self._familyVar.get().strip()

    #-- 2
    # [ if family is empty ->
    #   (display) += error popup
    #   self._familyEntry := self._familyEntry with focus
    # else ->
    #   self.teacher.given := value of self._givenVar
    #   self.teacher.middle := value of self._middleVar
    #   self.teacher.family := value of self._familyVar ]
    if family == '':
        tkMessageBox.showerror("Family name undefined",
                                "Please enter the instructor's names.")
        self._familyEntry.focus_set()
    else:
        self.teacher.given = self._givenVar.get().strip()
        self.teacher.middle = self._middleVar.get().strip()
        self.teacher.family = family
```

16.11. **ReqPanel._createControlFrame(): Add panel controls**

```python
def _createControlFrame(self):
    '''Create self._controlFrame and its widgets.
    ...'''
```
#-- 1
# [ self := self with a new ttk.Frame added
#   f := that ttk.Frame ]
f = ttk.Frame(self)

#-- 2
# [ f := f with a new ttk.Button showing REMOVE_INSTR_BUTTON
#   that removes self.teacher (if defined) from this
#   semester's requests, with a popup to allow the user
#   to cancel
#   self._removeInstrButton := that ttk.Button ]
rowx, colx = 0, 0
self._removeInstrButton = ttk.Button(f,
  text=REMOVE_INSTR_BUTTON,
  command=self._removeInstrHandler)
self._removeInstrButton.grid(row=rowx, column=colx,
  pady=5)

#-- 3
# [ f := f with a new ttk.Button showing CLOSE_BUTTON
#   that closes self, with a popup to allow the
#   user to cancel
#   self._closeButton := that ttk.Button ]
colx += 1
self._closeButton = ttk.Button(f,
  text=CLOSE_BUTTON,
  command=self.close)
self._closeButton.grid(row=rowx, column=colx,
  padx=10)

#-- 4
return f

16.12. ReqPanel._removeInstrHandler(): Handler for Remove this instructor from the CMS

- If the panel is not currently displaying one instructor's requests, that is, if self.teacher is None, the ReqPanel should not be onscreen. Otherwise we may assume that self.teacher currently contains a readadds.Teacher instance.
- If self.teacher has no Course children, we can just go ahead and delete it.
- If there are Course children, we'll need to show the user a popup that allows them to continue or cancel.

There is a subtle point about counting an instructor's sections and metacourses. The instructor could theoretically have a readadds.Course child which in turn had no readadds.Section children; that should not count as having any sections. Also, the instructor could have readadds.CrossParent children, but none of them have child sections. It's a good idea for the “delete section” functions in ReqPanel and MetaPanel to clean up empty courses and metacourses, respectively. However, just in case, we'll count the children and their children in both cases; these counts will be displayed in the error popup so the user knows how much data they are discarding.
def _removeInstrHandler(self):
    '''Handler for REMOVE_INSTR_BUTTON.

    [ self.teacher is a readadds.Teacher instance ->
      if (self.teacher has no Course children) or
      (the user clicks 'Yes' on a 'Discard child courses?' popup) ->
        self.app := self.app with self.teacher deleted
        from its current semester
    else -> ]
    ...

    #-- 1
    # [ sectionList := all the readadds.Section children of
    # those readadds.Course instances
    # metaList := all teacher's readadds.CrossParent children
    # metaChildList := all the readadds.Child children of
    # those readadds.CrossParent instances ]
    sectionList = [ section
                    for course in self.teacher.genCourses()
                    for section in course.genSections() ]
    metaList = [ crossParent
                 for crossParent in self.teacher.genCrosses() ]
    childList = [ child
                  for meta in metaList
                  for child in meta.genCRNs() ]

    #-- 2
    # [ if len(sectionList) + len(childList) == 0 ->
    #   I
    # else if the user selects 'Cancel' on an error popup ->
    #   return (as opposed to 'Discard') ->
    # else -> ]
    if len(sectionList) + len(childList) > 0:
        text = POP_INSTR_REQUEST_COUNT.format(
            who=self.teacher.fullName,
            uid=self.teacher.email,
            ns=len(sectionList),
            nx=len(metaList))
        reply = MultiDialog(self, POP_INSTR_NONEMPTY,
                            text,
                            [POP_INSTR_DISCARD, POP_INSTR_CANCEL])
        if reply.result == POP_INSTR_CANCEL:
            return

    #-- 2
    # [ self.app := self.app with self.teacher deleted
    #   from its current semester ]
    self.app.removeInstructor(self.teacher.email)
### 16.13. ReqPanel._createSectionsFrame(): Create the section list area

```python
# - - - ReqPanel._createSectionsFrame

def _createSectionsFrame(self):
    '''Create the widget group for showing teacher's sections.
    '''
    #-- 1
    # [ self := self with a new ttk.LabelFrame added
    #   f := that ttk.LabelFrame ]
    f = ttk.LabelFrame(self, text=SECTIONS_FRAME)
    #-- 2
    # [ f := f with a new SectionsList instance added that
    #   brings up the SectionPanel when the corresponding
    #   line is clicked
    #   self._sectionsList := that SectionsList instance ]
    rowx, colx = 0,0
    self._sectionsList = SectionsList(f,
        callback=self._sectionsListHandler)
    self._sectionsList.grid(row=rowx, column=colx,
        columnspan=3, padx=5, pady=5)
    #-- 3
    # [ f := f with a ttk.Entry added that brings up the
    #   SectionPanel transient when the user enters a
    #   CRN in self.app
    #   self._crnEntry := that ttk.Entry
    #   self._crnVar := the textvariable for that ttk.Entry ]
    rowx, colx = rowx+1, 0
    self._crnVar = tk.StringVar()
    self._crnEntry = ttk.Entry(f, width=6,
        font=self._entryFont,
        textvariable=self._crnVar)
    self._crnEntry.grid(row=rowx, column=colx, padx=5, pady=5,
        sticky=tk.E)
    self._crnEntry.bind("<KeyPress-Return>",
        self._crnHandler)
    #-- 4
    # [ f := f with a ttk.Button added that brings up the
    #   SectionPanel transient for the CRN in self._crnVar ]
    colx += 1
    self._findCrnButton = ttk.Button(f,
```

The listbox of sections is a compound widget described in Section 19, “class SectionsList: A scrollable listbox of section lines” (p. 119). The handler called when the user clicks on a section in this listbox is Section 16.14, “ReqPanel._sectionsListHandler(): Handler for clicking on a section in the sections list” (p. 107).

The handler called when the user presses the Enter key in self._crnEntry, or when they click on self._findCrnButton, is Section 16.15, “ReqPanel._crnHandler(): User entered a CRN to be added” (p. 108).
For the ttk.LabelFrame that encloses the pulldown window for bringing up a SchedPanel for a selected department, see Section 20, “class DeptSelector: Widget to bring up a department's schedule” (p. 125). Note that this widget spans two rows of the grid. Its handler is Section 16.17, “ReqPanel._deptHandler(): Show a department’s schedule in the swap area” (p. 110).

The handler for the Delete selected section button is Section 16.16, “ReqPanel._delSelSectionHandler: Handler for the Delete selected section button” (p. 109).

16.14. ReqPanel._sectionsListHandler(): Handler for clicking on a section in the sections list

A pass-through to Section 14.40, “App.showSection(): Bring up the SectionPanel” (p. 85).
transient that edits (section)
else -> I ]

#-- 1
self.app.showSection(section)

16.15. ReqPanel._crnHandler(): User entered a CRN to be added

This handler is both a command handler (for self._findCrnButton) and an even handler (for the Enter key pressed in self._crnEntry), which is why it has a generic calling sequence.

For the logic that checks for duplicate requests for the same CRN and adds the new section, see Section 16.2, “ReqPanel.addSection(): Add a section to the current instructor” (p. 96). That method needs a class_sched.SectionSchedule instance; we get that from Section 14.26, “App.lookupCRNSched(): Find a scheduled section by CRN” (p. 75).

cmsadds

# - - - R e q P a n e l . _ c r n H a n d l e r

# [ CRN_PATTERN := a re.MatchObject that matches a string
# containing exactly five digits ]
CRN_PATTERN = re.compile(r'\d{5}$')

def _crnHandler(self, *p):
    '''Look up a CRN and add its section (and course if necessary).'''
    #-- 1
    # [ if the value of self._crnVar, stripped, is exactly
    # five digits ->
    # crn := those five digits as a str
    # else ->
    # (display) += error popup
    # return ]
    crn = self._crnVar.get().strip()
    m = self.CRN_PATTERN.match(crn)
    if m is None:
        title = "Invalid CRN"
        text = ("A Course Reference Number must contain 
        "exactly five digits.")
        tkMessageBox.showerror(title, text)
        return

    #-- 2
    # [ if self.app has a scheduled section with CRN=(crn) ->
    # sectSched := the class_schedule.SectionSchedule
    # for that section
    # else ->
    # (display) += error popup
    # return ]
    try:
        sectSched = self.app.lookupCRNSched(crn)
    except KeyError:
        title = "Unknown CRN"
On a successful search, we not only add the section to the sections list, but also clear the CRN field so the operator doesn’t have to clear it before entering another CRN.

```
#-- 3
# [ if there is an existing request for CRN (sectSched.crn)
# in self.app ->
# (display) +=: a popup showing what teacher has
# that request
# else ->
# self._crnVar := self._crnVar, cleared
# self.teacher := self.teacher with a new
# readadds.Section instance added, including its
# parent readadds.Course instance if necessary ]
self._crnVar.set('')
self.addSection(sectSched)
```

16.16. ReqPanel._delSelSectionHandler: Handler for the Delete selected section button

There are two possible errors associated with the Delete selected section button.

- The user forgot to select a section to be deleted.
- The selected section is a member of a cross-listed set for this instructor, so deleting it would cause an integrity failure. The readadds.Course.delete() method will raise readadds.IntegrityError if that is the case.

```
#- - - R e q P a n e l . _ d e l S e l S e c t i o n H a n d l e r

def _delSelSectionHandler(self):
    '''Handler for DEL_SEL_SECTION.

    [ if (no section is selected in self._sectionsList) or
      (deleting that section from its course would cause an
      integrity failure) ->
      (display) +=: an error popup
      return
    else if transient-can-close ->
      self.app := self.app with no transient
      that course := that course - (that section) ]
    ...
#- - 1
# [ if a section is selected in self._sectionsList ->
#    # killSection := that section as a readadds.Section
#    # parentCourse := parent of that section as a
#    # readadds.Course
#    # else ->
```
killSection = self._sectionsList.get()
if killSection is None:
    title = "No section is selected"
    text = ("To delete a section request, first click on "
             "its line in the list, then click the "
             "Delete selected section button.")
    tkMessageBox.showerror(title, text)
    return
else:
    parentCourse = killSection.course
    We must close the transient panel because it might be displaying the section about to be deleted.

try:
    parentCourse.delete(killSection)
    self._sectionsList.delete(killSection)
except readadds.IntegrityError as x:
    title = ("Can't delete section {s.crn}, "
             "{s.fullCode}".format(s=killSection))
    text = str(x)
    tkMessageBox.showerror(title, text)
    return

16.17. ReqPanel._deptHandler(): Show a department’s schedule in the swap area

This method is the callback from Section 20, “class DeptSelector: Widget to bring up a department’s schedule” (p. 125); its argument is a readadds.DeptSchedule instance.

For a given semester, there may be no scheduled sections in that department. In that case, the user gets an error popup.
def _deptHandler(self, deptCode):
    '''Handler for a click on self._deptSelector.
    [ deptCode is a str ->
    if deptCode is a department code in app.semSched ->
    app := app displaying the corresponding DeptSchedule instance
    else ->
    (display) := an error popup ]
    ...
    #-- 1
    # [ (app.semSched is not None) ->
    #  if deptCode is a department code in the schedule for
    #  app.semSched ->
    #  if transient-can-close ->
    #  self := self displaying the SchedPanel for
    #  the department with code (deptCode)
    #  else -> raise CloseFailure
    #  else ->
    #  (display) += error popup: no classes scheduled
    #  in dept (deptCode) for semester self.semSched ]
    self.app.showDept(deptCode)

16.18. ReqPanel._createMetaFrame(): Create the metacourse widget group

def _createMetaFrame(self):
    '''Create the widgets to manage an instructor's metacourses.''
    ...
    #-- 1
    # [ self := self with a new ttk.Frame
    #  f := that ttk.Frame ]
    f = ttk.LabelFrame(self, text=META_FRAME)
    rowx, colx = 0, 0
    self._addMetaButton = ttk.Button(f,
        text=ADD_META_BUTTON,
        command=self._addMetaHandler)
    self._addMetaButton.grid(row=rowx, column=colx)
The handler for the `Delete selected metacourse` button is Section 16.20, “ReqPanel._delSelMetaHandler(): Handler for the `Delete selected metacourse` button” (p. 113).

```python
#-- 3
# [ f := f with a new ttk.Button added that deletes
#   the selected metacourse
#   self._delSelMetaButton := that ttk.Button ]
colx += 1
self._delSelMetaButton = ttk.Button(f,
    text=DEL_SEL_META_BUTTON,
    command=self._delSelMetaHandler)
self._delSelMetaButton.grid(row=rowx, column=colx,
                           padx=5)
```

The listbox showing the current set of metacourses for this instructor is an instance of Section 21, “class MetaList: Listbox showing the instructor’s metacourses” (p. 131). For the logic that brings up a metacourse in the MetaPanel, see Section 16.21, “ReqPanel._editMetaHandler(): Bring up a metacourse for editing” (p. 114).

```python
#-- 4
# [ f := f with a new MetaList added
#   self._metaList := that MetaList ]
rowx, colx = rowx+1, 0
self._metaList = MetaList(f, callback=self._editMetaHandler)
self._metaList.grid(row=rowx, column=colx,
                      columnspan=2, pady=5)
```

**16.19. ReqPanel._addMetaHandler(): Handler for the `Add metacourse` button**

This button adds a new `readadds.CrossParent (metacourse)` instance to `self.teacher` and to `self._metaList`, then brings up the MetaPanel transient to display it for editing.

```python
#-- 5
return f
def _addMetaHandler(self):
    # 'Add a new metacourse.
    # [ self.teacher is a readadds.Teacher instance ->
    #   if transient-can-close ->
    #     self.teacher := a new readadds.CrossParent instance
    #       with an empty title and no child sections
    #     self.app := self.app displaying that new
    #       instance in the MetaPanel transient
    #   else -> I ]
    ...  
```
16.20. **ReqPanel._delSelMetaHandler()**: Handler for the *Delete selected metacourse* button

Now if we are going to delete a metacourse, consider three cases based on what is currently displayed in the transient area.

- If the swap area is clear, there is no problem.
- If the swap area does not currently show a MetaPanel, the deletion of a metacourse doesn't really affect it, so we should leave it there: otherwise the user might be annoyed at having to bring back their SchedPanel or SectionPanel.
- The complicated case is when there is a MetaPanel currently showing in the swap area. There are two subcases: if the metacourse currently showing there is not the one the user is asking us to delete, again we don't need to erase it.

This leaves the case where there's a MetaPanel showing in the swap area that is displaying the metacourse to be deleted. We can't use Section 14.29, “App.closeTransient: Clear the swap area” (p. 78) to get rid of it, because it might very well be in an invalid state, and it would ask the user if they want to discard it. However, since we got here only because the user clicked *Delete selected metacourse*, that shouldn't be necessary.

Hence, we need a method in the App class with very specific semantics: if the swap area contains a MetaPanel that is displaying the metacourse we are trying to delete, kill it without mercy. That method is Section 14.44, “App.clearSpecificMeta()” (p. 87).

To delete the metacourse that is selected in self._metaList, the first thing we check for is that there is a selected entry; see Section 21.5, “MetaList.getSelected(): Which metacourse is selected?” (p. 134).

```python
# - - -   R e q P a n e l . _ d e l S e l M e t a H a n d l e r

def _delSelMetaHandler(self):
```

```python
# if transient-can-close ->
# self.app := self.app showing no transient
# else -> return ]
try:
    self.app.closeTransient()
except CloseFailure:
    return

#-- 2
# [ self.teacher += a new readadds.CrossParent instance
#     with an empty description
#     crossParent := that readadds.CrossParent instance
#     self._metaList += that readadds.CrossParent instance ]
crossParent = readadds.CrossParent(self.teacher, '')
self._metaList.add(crossParent)

#-- 3
# [ self.app := self.app showing a MetaPanel that edits
crossParent ]
self.app.showMeta(crossParent)
```
Handler for DEL_SEL_META_BUTTON.

#-- 1
# [ if a line in self._metaList is selected ->
#   crossParent := the corresponding readadds.CrossParent
# else ->
#   (display) += error popup
# return ]
crossParent = self._metaList.getSelected()
if crossParent is None:
    title = "No metacourse is selected"
    text = ("To delete a metacourse, select it in the "
            "list below, then click '{0}'".format(DDEL_SEL_META_BUTTON))
tkMessageBox.showerror(title, text)
return

#-- 2
# [ if self.app's swap area is displaying crossParent ->
#   swap area := swap area, cleared
# else -> I ]
self.app.clearSpecificMeta(crossParent)

Now we can delete the metacourse from self._metaList; see Section 21.3, "MetaList.delete(): Remove a cross-listed set" (p. 133). Actual deletion of the readadds.CrossParent from its parent is performed by the et.Element.remove() method that readadds.Teacher inherits.

#-- 3
# [ self._metaList := self._metaList without crossParent
#   self._teacher := self._teacher without crossParent ]
self._metaList.delete(crossParent)
self.teacher.remove(crossParent)

16.21. ReqPanel._editMetaHandler(): Bring up a metacourse for editing

This is the handler invoked when the user right-clicks on a line in self._metaList. It brings up the selected metacourse in the MetaPanel.
try:
    self.app.closeTransient()
except CloseFailure:
    return

#-- 2
# [ self.app := self.app displaying the MetaPanel editing (crossParent)
#    self.app.showMeta(crossParent)

17. class ScrolledList: Generic listbox with one or two scrollbars

This class is a combination widget of a `tk.Listbox`, always with a vertical scrollbar, optionally also with a horizontal scrollbar. It exports its internal `tk.Listbox` widget as its `.list` attribute.

cmsadds

# - - - - - c l a s s S c r o l l e d L i s t
class ScrolledList(ttk.Frame):
    '''Generic scrollable list.

    Exports:
    ScrolledList(parent, horiz=False, frameArgs=None,
                 listboxArgs=None):
        [ (parent is a ttk.Frame) and
          (horiz is a bool) and
          (frameArgs is a dict or None) and
          (listboxArgs is a dict or None) ->
          parent := parent with a new ScrolledList added, using
          frameArgs as its options, containing a tk.Listbox
          using listboxArgs is its options, with a vertical
          scrollbar, and a horizontal scrollbar iff (horiz)
          return that new ScrolledList
          .list: [ self's tk.Listbox ]
        ]

    Internal widgets:
    .yScroll: [ vertical ttk.Scrollbar ]
    .xScroll: [ horizontal ttk.Scrollbar if requested ]
    ...

    def __init__(self, parent, horiz=False, frameArgs=None,
                 listboxArgs=None):
        '''Constructor.
        ...'

    #-- 1
    # [ frameDict := frameArgs defaulting to {}
    #    listboxDict := listboxArgs defaulting to {}
    frameDict = (frameArgs
                 if frameArgs is not None
                 else {}}

cmsadds: A GUI for CMS course creation
listboxDict = (listboxArgs
if listboxArgs is not None
else {})

#-- 2
# [ parent := parent with a new ttk.Frame with options
#   from frameArgs if given
ttk.Frame.__init__(self, parent, **frameDict)
self.grid()

#-- 3
# [ self := self with a new tk.Listbox added with options
#   from listboxArgs if given
# self.list := that tk.Listbox ]
self.list = tk.Listbox(self, **listboxDict)
self.list.grid(row=0, column=0, sticky=tk.N+tk.S+tk.E+tk.W)

#-- 4
# [ self := self with a new vertical ttk.Scrollbar added
#   linked to self.list
# self._yScroll := that ttk.Scrollbar ]
self._yScroll = ttk.Scrollbar(self, orient=tk.VERTICAL,
command=self.list.yview)
self._yScroll.grid(row=0, column=1, sticky=tk.N+tk.S)
self.list['yscrollcommand'] = self._yScroll.set

#-- 5
# [ if horiz ->
#   self := self with a new horizontal ttk.Scrollbar added
#   linked to self.list
# self._xScroll := that ttk.Scrollbar
# else -> I ]
if horiz:
    self._xScroll = ttk.Scrollbar(self, orient=tk.HORIZONTAL,
command=self.list.xview)
    self._xScroll.grid(row=1, column=0, sticky=tk.E+tk.W)
    self.list['xscrollcommand'] = self._xScroll.set

18. class Pager: Base widget with swappable content

This class implements the behavior of two areas in the application that have content that can change or vanish. The ReqPanel vanishes when there is no instructor selected; and the SwapArea has multiple transients, and can also be blank.

To use this widget:

1. Create and grid the Pager instance $P$.

2. For each child frame $C$, create $C$ as the child of $P$, then register it by:

   $P.add(C)$

3. To make child $C$ appear, whether the window is currently blank or displaying a different child:
4. To blank the window:

```python
P.clear()
```

```python
# - - - - - class Pager
class Pager(ttk.Frame):
    '''Page-changer widget.

    Exports:
    Pager(parent, size=None, **kw):
        [ (parent is a frame) and
          (size is a requested size as a tuple (width, height),
           defaulting to automatic sizing to contents) and
          (kw is a dict with Frame options) ->
          parent := parent with a new, unregistered Pager
          frame widget added, using kw to create its frame,
          with no content
          return that new Pager ]
    .parent: [ as passed to constructor ]
    .add(w):
        [ w is a widget whose parent is self.parent ->
          self := self with w added to its collection ]
    .select(w):
        [ w is a widget in self ->
          self := self displaying w ]
    .clear():
        [ self := self not displaying any widget ]
    .get():
        [ if self is displaying a child widget ->
          return that widget
          else -> return None ]

    Private attributes include a set of the registered children and an attribute that remembers which child
    widget, if any, is displayed.

    State/Invariants:
    ._childSet:
        [ a set containing self's child widgets ]
    ._current:
        [ if self is not displaying anything -> None
          else -> the member of self._childSet currently showing ]
   ..
```
18.1. Pager.add(): Register a child frame

```python
# - - -  P a g e r . a d d

@adddef add(self, w):
    '''Add w to self's content set.
    '''
    self._childSet.add(w)
```

18.2. Pager.select(): Make a child appear

```python
# - - -  P a g e r . s e l e c t

selectdef select(self, w):
    '''Display w
    '''
    #-- 1
    # [ if self._current is not None ->
    #   self := self with self._current removed
    #   self._current := None
    # else -> I ]
    self.clear()
    #-- 2
    # [ w is in self._childSet ->
    #   self := self with w displayed ]
    if w not in self._childSet:
        raise ValueError("Pager.select(): This widget has not 
        "
        "been registered with Pager.add().")
    w.grid()
    self._current = w
```

18.3. Pager.clear(): Erase the window

```python
# - - -  P a g e r . c l e a r

cleardef clear(self):
    '''If content is displayed, remove it.
    '''
    #-- 1
    # [ if self._current is not None ->
    #   self := self with self._current removed
    #   self._current := None
    # else -> I ]
    if self._current is not None:
        self._current.grid_remove()
        self._current = None
```
18.4. Pager.get(): Return the currently displayed widget

```python
# --- Pager.get
def get(self):
    '''Return the currently displayed widget or None
    '''
    return self._current
```

18.5. Pager.__init__()

```python
# --- Pager.__init__
def __init__(self, parent, size=None, **kw):
    '''Constructor.
    '''
    #-- 1
    # [ self := self as a ttk.Frame with kw as its options,
    #    unregistered ]
    ttk.Frame.__init__(self, parent, **kw)
    
    #-- 2
    # [ if size is not None ->
    #   self := self with fixed width size[0] and height size[1]
    # else -> I ]
    if size is not None:
        w, h = size
        self.config(width=w, height=h)
        self.grid_propagate(0)

    #-- 2
    self.size = size
    self._childSet = set()
    self._current = None
```

19. class SectionsList: A scrollable listbox of section lines

This widget appears in multiple places, wherever a set of sections needs to be managed. It is derived from the generic scrolled list type, Section 17, “class ScrolledList: Generic listbox with one or two scrollbars” (p. 115).

```python
# --- --- class SectionsList
```
class SectionsList(ScrolledList):
    '''Scrollable list of sections.

    Exports:
    SectionsList(parent, callback=None):
        [ parent := parent with a new SectionsList child added
          displaying no sections, and calling callback(s)
          when a line is clicked, where s is an instance of
          readadds.Section ]
    .clear():
        [ self := self blank and with no child sections ]
    .add(section):
        [ section is a readadds.Section instance ->
          if self contains section ->
            I
          else ->
            self := self with a line added at the end displaying
              section ]
    .nSections(): [ return the number of sections in self ]
    .genSections():
        [ generate the sections in self as a sequence of
          readadds.Section instances ]
    .delete(section):
        [ section is a readadds.Section ->
          if self has an entry for section ->
            self := self with that entry removed
          else -> I ]
    .get():
        [ if a line in self is selected ->
          return the readadds.Section associated with that line
          else -> return None ]

    State/Invariants:
    ._sectList:
        [ a list containing self's current readadds.Section instances,
          in the same order they are displayed in the listbox ]

    ...
To implement the callback, we bind a handler to a click on button 1 to Section 19.10, “SectionsList._clickHandler(): Handler for clicking on an item” (p. 124).

```python
#-- 2
self.callback = callback
self._sectList = []
```

#-- 3
```python
# [ self.list := self.list with a binding that calls
#   self._clickHandler on even <Button-3> ]
self.list.bind("<Button-3>", self._clickHandler)
```

### 19.2. SectionsList.clear()

```python
#--- SectionList.clear

def clear(self):
    '''Make self empty inside and out.
    '''
    #-- 1
    self._sectList = []
    self.list.delete(0, tk.END)
```

### 19.3. SectionsList.add(): Add a new section

```python
#--- SectionsList.add

def add(self, section):
    '''Add a new read.Adds.Section instance to self.
    '''
    #-- 1
    if section in self._sectList:
        return
    else:
        self._sectList.append(section)
```

Rather than trying to modify the existing Listbox, we just rebuild it after any change; see Section 19.4, “SectionsList.rebuild(): Rebuild the displayed list” (p. 122).

```python
#-- 2
# [ self.list := self.list rebuilt using self._sectList ]
self._rebuild()
```
19.4. SectionsList.rebuild(): Rebuild the displayed list

```python
#--- SectionsList._rebuild
def _rebuild(self):
    '''Rewrite the entire listbox.
    [ self.list := self.list rebuilt using self._sectList ]
    '''
    #-- 1
    # [ self.list := self.list cleared ]
    self.list.delete(0, tk.END)
```

The logic that formats the lines is Section 19.5, “SectionsList._showLine(): Format a line displaying a SectionSchedule” (p. 122).

```python
#-- 2
# [ self.list += lines displaying the elements of
# for section in self._sectList:
#     self.list.insert(tk.END, self._showLine(section))
```

19.5. SectionsList._showLine(): Format a line displaying a Section-Schedule

```python
#--- SectionsList._showLine
def _showLine(self, section):
    '''Return a line displaying information from section.
    [ section is a readadds.Section instance ->
      return a str showing the CRN, department code, course number, section type, section number, and title from
      section ]
    '''
    #-- 1
    return ("{crn} {code} {title}".format(
        crn=section.crn,
        code=section.fullCode,
        title=section.desc))
```

19.6. SectionsList.genSections(): Generate the sections

```python
#--- SectionsList.genSections
def genSections(self):
    '''Generate the contained readadds.Section instances.
    '''
```
for section in self._sectList:
    yield section

19.7. SectionsList.delete()

Deletes one of the lines in the box. The list is rebuilt after modification; see Section 19.4, “SectionsList.rebuild(): Rebuild the displayed list” (p. 122).

```python
# - - - S e c t i o n s L i s t . d e l e t e

def delete(self, section):
    '''Delete a readaddons.Section from the list.
    '''
    #-- 1
    if section in self._sectList:
        self._sectList.remove(section)
    #-- 2
    self._rebuild()
```

19.8. SectionsList.get(): Retrieve the currently selected SectionSchedule

This method assumes that the tk.Listbox allows only one line at a time to be selected, that is, it was build with the selectmode option tk.BROWSE (the default) or tk.SINGLE. If multiple items are se-
lected, this will return only the first.

```python
# - - - S e c t i o n s L i s t . g e t

def get(self):
    '''Retrieve the currently selected SectionSchedule
    '''
    #-- 1
    selTuple = self.list.curselection()
    if len(selTuple) == 0:
        return None
    else:
        listx = int(selTuple[0])
```
19.9. SectionsList.nSections(): How many child sections are there?

```python
# - - - S e c t i o n s L i s t . n S e c t i o n s

def nSections(self):
    '''Return the number of child sections.
    '''
    return len(self._sectList)
```

19.10. SectionsList._clickHandler(): Handler for clicking on an item

```python
# - - - S e c t i o n s L i s t . _ c l i c k H a n d l e r

def _clickHandler(self, event):
    '''Respond to a Button-3 click.
    [ if self.callback is None ->
    I
    else ->
    call self.callback(s) where s is the readadds.Section
    corresponding to the currently selected line in
    self.list ]
    '''
    #-- 1
    # [ if self.callback is None ->
    #  return
    #  else ->
    #    linex := index of the line in self.list closest
    #    to event.y
    #    xmin := minimum X coordinate of that line
    #    xmax := maximum X coordinate of that line
    #    ymin := minimum Y coordinate of that line
    #    ymax := maximum Y coordinate of that line ]
    if self.callback is None:
        return
    else:
        linex = self.list.nearest(event.y)
        xmin, ymin, wide, high = self.list.bbox(linex)
        ymax = ymin + high - 1
        if ymin <= event.y <= ymax:
            self.callback(self._sectList[linex])
```

If the user clicks near but not inside the bounding box of line number `linex`, we should ignore it; otherwise we call the callback and pass it the `readadds.Section` instance corresponding to line `linex`.

```python
#-- 2
```
**Note**

The first version of the above bit checked both $x$ and $y$ values to see if they were in the bounding box. However, this required that the user click within the area of the line actually occupied by text, so if they clicked to the right of the last character, it wouldn't respond. Hence our practice of checking only the $y$ value.

Since writing the above paragraph, the author has discovered that you won't get an event if the user clicks past the end of the text anyway.

### 20. class DeptSelector: Widget to bring up a department's schedule

```python
# - - - - - class DeptSelector
class DeptSelector(ttk.LabelFrame):
   '''Contains a drop-down combobox to select a department.

Exports:
   DeptSelector(parent, app, callback=None):
      [ (parent is a ttk.Frame) and
        (app is the containing App instance) and
        (callback is a function to be called with a department
        code when selected, or None) ->
         parent := parent with a new DeptSelector added
displaying the department codes in app.classSchedule
        return that new DeptSelector ]
   .parent: [ as passed to constructor, read-only ]
   .app: [ as passed to constructor, read-only ]
   .callback: [ as passed to constructor, read-only ]
   .select(deptCode):
      [ deptCode is a department code in self ->
        self := self showing that department ]

State/Invariants:
   ._deptCode: [ the currently selected department code ]
   ._deptCodeList:
      [ a list of all the department codes in self, in
      ascending order by department name ]
   ._deptCodeMap:
      [ a dict whose keys are the elements of self._deptCodeList
      and each related value is the related department name ]
   ._entryFont: [ a tkFont.Font ]

Widgets:
   ._deptCodeEntry:
      [ a ttk.Entry where the user can enter a department code ]
   ._deptCodeVar:
```

---

*New Mexico Tech Computer Center*  
*cmsadds: A GUI for CMS course creation*  
125
20.1. DeptSelector.select(): Display a given department code and name

The select function displays the department code and name in the Menubutton. It also clears the department code entry.

```python
# - - - DeptSelector.select

def select(self, deptCode):
    '''Display the selected department.''
    text = u"{0}: {1}".format(deptCode, self.app.lookupDeptName(deptCode))
    self._deptCode = deptCode
    self._mbVar.set(text)
    self._deptCodeVar.set('')
```

20.2. DeptSelector.__init__()

```python
# - - - DeptSelector.__init__

def __init__(self, parent, app, callback=None):
    '''Constructor.''
    #-- 1
    # [ self := self as a ttk.LabelFrame, registered ]
    ttk.LabelFrame.__init__(self, parent, text=SHOW_DEPT_SCHED_FRAME)

    #-- 2
    self.parent = parent
    self.app = app
    self.callback = callback
```
self._entryFont = tkFont.Font(family=MONOSPACE_FAMILY, size=18)

#-- 3
# [ self._deptCodeList := as invariant
#    self._deptCodeMap := as invariant ]
self._buildList()

#-- 4
# [ self := self with all subsidiary widgets added ]
self._createWidgets()

#-- 5
# [ self := self with the first element of self._deptList
#    selected ]
self.select(self._deptCodeList[0])

20.3. DeptSelector._buildList(): Build the list of departments

To make departments easier to find in the menu, we alphabetize them by department name, not code. We'll need app.classSchedule to translate codes to names. First we build the list in whatever order the departments come out of app.semSched.genDepts().

```python
# - - - D e p t S e l e c t o r . _ b u i l d L i s t
def _buildList(self):
    '''Build the list of department schedules.
    '''
    #-- 1
    # [ self._deptList := a list of department codes in
    #    app.classSchedule ]
    self._deptCodeList = [deptCode
                          for deptCode in self.app.classSchedule.genDeptCodes()]

    Now build self._deptCodeMap so we can look up these instances by (uppercased) department code.

    #-- 2
    # [ self._deptCodeMap := as invariant ]
    self._deptCodeMap = {deptCode: self.app.lookupDeptName(deptCode)
                         for deptCode in self._deptCodeList}

    For the comparator used to sort the departments by name, see Section 20.4, “DeptSelector._byDeptName(): Sort departments by name” (p. 128).

    #-- 3
    # [ self._deptCodeList := self._deptCodeList sorted by
    #    department name ]
    self._deptCodeList.sort(cmp=self._byDeptName)
```
20.4. DeptSelector._byDeptName(): Sort departments by name

This comparator orders DeptSchedule instances according to their name. These instances carry only the department code; we must translate them using app.classSchedule.

```python
# --- DeptSelector._byDeptName

def _byDeptName(self, code1, code2):
    '''Orders departments by their names.

        [ (self.deptCodeMap is as invariant) and
          (code1 and code2 are department codes in self) ->
          return cmp(code1's name, code2's name), case-insensitive ]
    ...
    #-- 1
    name1 = self.deptCodeMap[code1].upper()
    name2 = self.deptCodeMap[code2].upper()
    #-- 2
    return cmp(name1, name2)
```

20.5. DeptSelector._createWidgets()

The handler for either the Enter key in the department code field, or clicking on the adjacent Find department button, is Section 20.7, “DeptSelector._findDeptHandler(): Handler to look up a department code” (p. 130).

```python
# --- DeptSelector._createWidgets

def _createWidgets(self):
    '''Create all widgets.
    ...
    #-- 1
    # [ self := self with a new ttk.Entry that selects
    #   a department by code
    # self.deptCodeVar := as invariant ]
    # self.deptCodeEntry := that ttk.Entry
    rowx, colx = 0, 0
    self.deptCodeVar = tk.StringVar()
    self.deptCodeEntry = ttk.Entry(self,
        font=self._entryFont,
        width=6,
        textvariable=self.deptCodeVar)
    self.deptCodeEntry.grid(row=rowx, column=colx,
        padx=5, pady=5, sticky=tk.E)
    self.deptCodeEntry.bind("<KeyPress-Return>",
        self._findDeptHandler)
    #-- 2
    # [ self := self with a new ttk.Button that calls
    #   self._findDeptHandler
    # self.deptFindButton := that ttk.Button ]
```
Next we create the Menubutton and its associated Menu. Each command on the menu selects the corresponding department. At this writing (June 2013), the longest department title is that of CSE.

```python
rowx, colx = rowx+1, 0
self._mbVar = tk.StringVar()
self._mb = ttk.Menubutton(self,
    textvariable=self._mbVar,
    width=len("CSE: Computer Science + Engineering"))
self._mb.grid(row=rowx, column=colx, columnspan=3,
    padx=5, pady=5)
self._menu = tk.Menu(self._mb, tearoff=0)
self._mb['menu'] = self._menu
```

For the logic that builds the set of commands in the menu, see Section 20.6, “DeptSelector._addDept(): Add one department command” (p. 129).

```python
for deptCode in self._deptCodeList:
    #-- 4 body
    # [ self._menu := a command showing the department code
    #    and name for deptCode that selects deptCode ]
    self._addDept(deptCode)
```

---

### 20.6. DeptSelector._addDept(): Add one department command

We’ll build a closure to use as the command. See Section 20.8, “DeptSelector._show(): Display the schedule for the current department” (p. 131).

```python
#-- - - D e p t S e l e c t o r . _ a d d D e p t
def _addDept(self, deptCode):
    '''Add a department command to self's menu.
    '''
    #-- 1
    # [ handler := a function that shows (deptCode) ]
```
def handler():
    '''Handler for change of department.
    '''
    #-- 1
    # [ self := self showing deptCode ]
    self.select(deptCode)
    #-- 2
    # [ if self.callback is not None ->
    #     call self.callback(self._deptCode)
    # else -> I ]
    self._show()

Now we'll build the command. The label has the form “dept code: dept name”.

#-- 2
# [ self._menu += a new command displaying semSched ]
text = "{0}: {1}".format(deptCode,
                          self.app.lookupDeptName(deptCode))
self._menu.add_command(command=handler, label=text)

20.7. DeptSelector._findDeptHandler(): Handler to look up a department code

This is the handler for the Find department button. If the contents of self._deptCodeVar match one of the department codes, that department is selected in the menu. Because it is called both through the command linkage and as an event handler, it uses a generic calling sequence.

def _findDeptHandler(self, *p):
    '''Look up the department code and try to select it.
    
    [ if the contents of self._deptCodeVar match one of the
department codes in self._deptList ->
    self := self displaying that department
    else ->
    (display) := an error popup ]
    ...'''
    #-- 1
    # [ deptCode := contents of self._deptCodeVar, uppercased ]
deptCode = self._deptCodeVar.get().upper()
    #-- 2
    # [ deptCode is a key in self._deptCodeMap ->
    #    deptSched := the related value
    # else ->
    #    (display) := an error popup
    # return ]
    if deptCode not in self._deptCodeMap:
        tkMessageBox.showerror("Invalid department code",
                           "Department code '{0}' has no scheduled courses for ")
20.8. DeptSelector._show(): Display the schedule for the current department

```python
def _show(self):
    '''Call the callback if there is one.
    '''
    #-- 1
    # [ if self.callback is not None ->
    #     call self.callback(deptCode)
    # else -> I ]
    if self.callback is not None:
        self.callback(self._deptCode)
```

21. class MetaList: Listbox showing the instructor's metacourses

This class inherits from Section 17, “class ScrolledList: Generic listbox with one or two scrollbars” (p. 115).

```python
class MetaList(ScrolledList):
    '''Listbox showing the instructor's metacourses.
    Exports:
    MetaList(parent, callback=f):
        [ (parent is a ttk.Frame) and
          (f is a function that will be called with one
          readadds.CrossParent instance when the line
          for that CrossParent is right-clicked) ->
          parent := parent with a new MetaList added
          return that new MetaList ]
```
21.1. MetaList.add(): Add a cross-listed set

```python
# - - - MetaList.add

def add(self, crossParent):
    '''Add a cross-listed set to self.
    ...'''
    #-- 1
    if crossParent in self._crossList:
        return
    else:
        self._crossList.append(crossParent)
```

Rather than modify the listbox in place, we just rebuild it after any change; see Section 21.2, “MetaList._rebuild(): Rebuilt the listbox” (p. 132).

```python
#-- 2
self._rebuild()
```

21.2. MetaList._rebuild(): Rebuilt the listbox

```python
# - - - MetaList._rebuild

def _rebuild(self):
    '''Erase and rebuild the listbox.
    ...'''
```
21.3. MetaList.delete(): Remove a cross-listed set

```python
# -- 1
# [ self.list := self.list, empty ]
self.list.delete(0, tk.END)

# -- 2
# [ self.list := self.list with a line added showing
#   the title of each element self._crossList ]
for cross in self._crossList:
    self.list.insert(tk.END, cross.desc)
```

Removing a readadds.CrossParent from a readadds.Teacher is actually performed by the latter’s inherited et.Element.remove() method. Also see Section 21.2, “MetaList._rebuild(): Rebuilt the listbox” (p. 132).

```python
# -- 2
# [ self._crossList := self._crossList with crossParent
#   deleted ]
self._crossList.remove(crossParent)

# -- 3
# [ self := self erased and rebuilt ]
self._rebuild()
```

21.4. MetaList.clear(): Clear the list

```python
# -- - - - MetaList.clear

def clear(self):
    '''Make the list empty.
    '''
    self.list.delete(0, tk.END)
    self._crossList = []
```
21.5. MetaList.getSelected(): Which metacourse is selected?

```python
# MetaList.getSelected

def getSelected(self):
    '''Return the selected readads.CrossParent or None.
    '''
    #-- 1
    # [ selTuple := tuple of line numbers of selected lines
    # in self.list, as strings ]
    selTuple = self.list.curselection()
    #-- 2
    if len(selTuple) == 1:
        return self._crossList[int(selTuple[0])]
    else:
        return None
```

21.6. MetaList.__init__()

```python
# MetaList.__init__

```def __init__(self, parent, callback=None):
    '''Constructor.
    '''
    #-- 1
    # [ self := self as a ScrolledList, calling
    # self._clickHandler on right-click ]
    ScrolledList.__init__(self, parent, horiz=True,
                         listboxArgs=dict(height=4, width=40))
    self.list.bind("<Button-3>", self._clickHandler)
    #-- 2
    self.parent = parent
    self.callback = callback
    self._crossList = []
```

21.7. MetaList._clickHandler(): Handle right-click on the metacourse list

```python
# MetaList._clickHandler

def _clickHandler(self, event):
    '''Handle click on a metacourse.
    [ event is a tk.Event ->
      if (self.callback is None) or (event's click is
      not on a line in self) ->
      I
```
else ->
    call self.callback(the readadds.CrossParent corresponding to the click location) ]

...  

#-- 1
if self.callback is None:
    return

#-- 2
# [ nearx := the index of the visible line in self
#   closest to event.y ]
nearx = self.list.nearest(event.y)

#-- 3
# [ if event.y is not inside the bounding box of line
#   (nearx) ->
#   return
#   else ->
#       call self.callback(self._crossList[nearx]) ]
xmin, ymin, wide, high = self.list.bbox(nearx)
ymax = ymin + high - 1
if ymin <= event.y <= ymax:
    self.callback(self._crossList[nearx])

22. **class SwapArea**: Home for transient panels

This compound widget is a space occupied by other compound widgets. It handles allocation of that space, and knows which child widget (if any) is currently displayed.

```python
#-- class SwapArea

class SwapArea(Pager):
    '''Swap area for transient panels.

    Exports:
    SwapArea(parent, **kw):
        [ (parent is a ttk.Frame) and
          (kw is a dict of ttk.Frame options) ->
          parent += a new ttk.Frame with options from kw
          return a new SwapArea instance with no child
          panel displayed ]

    See Section 10.12, “transient-can-close: Can the transient panel be removed?” (p. 41).

    .close():
        [ if transient-can-close ->
          self := self not displaying a transient
          return None
        else -> raise CloseFailure ]
```

New Mexico Tech Computer Center  cmsadds: A GUI for CMS course creation
22.1. SwapArea.__init__()

def __init__(self, parent, **kw):
    r'''Constructor.
    '''
    #-- 1
    # [ self := a new ttk.Frame with options from kw ]
    Pager.__init__(self, parent, size=(SWAP_WIDE, SWAP_HIGH),
                   borderwidth=5, relief=tk.GROOVE, **kw)

22.2. SwapArea.close(): Blank the swap area

def close(self):
    r'''Blank the swap area, if the current transient can close
    '''
    #-- 1
    if self._current is None:
        return
    #-- 2
    # [ if (self._current's contents are not in a valid
    #   state) and
    #   (the user elects to leave self._current onscreen) ->
    #   raise CloseFailure
    # else -> I ]
    self._current.close()

    #-- 3
    self.clear()

23. class Transient(): Base class for panels in the swap area

Panels that appear in the swap area will inherit from this class. The generic intended function for the
.close() method is declared at this level; the verb “remove” in its intended function is used to mean
the semantics of the Tkinter .grid_remove() function, which makes the frame disappear from the
screen but retains its attributes so that it can be made to reappear through a call to .grid().

class Transient(ttk.Frame):
    r'''Base class for swap area panels.

    Exports:
    Transient(swapArea, **kw):
    '''
23.1. Transient.__init__()

The constructor calls its parent constructor and stores the swapArea argument.

```python
# - - - T r a n s i e n t . _ _ i n i t _ _

def __init__(self, swapArea, **kw):
    """Constructor."
    ...
    #-- 1
    # [ swapArea := swapArea with a new ttk.Frame added using
    # attributes from (kw)
    # self := that ttk.Frame ]
    ttk.Frame.__init__(self, swapArea, **kw)
    #-- 2
    self.swapArea = swapArea
```

23.2. Transient.close(): Default close function

For panels that are don't have an invalid state (for example DeptPanel, which is read-only), this default .close() method removes itself from the parent and does not raise a CloseFailure exception.

```python
# - - - T r a n s i e n t . c l o s e

def close(self):
    """Remove self and indicate success."
    ...
    #-- 1
    # [ self.swapArea := self.swapArea with self removed ]
    self.swapArea.clear()
```
24. **class SchedPanel: Display all the sections in a given department**

This class represents one of the panels that can appear in the swap area, so it inherits from Section 23, "class Transient(): Base class for panels in the swap area" (p. 136).

```python
# - - - - - c l a s s  S c h e d P a n e l
class SchedPanel(Transient):
    '''Department schedule display panel.

Exports:
SchedPanel(swapArea, callback):
    [ (swapArea is a ttk.Frame) and
      (callback is a function that will be called when the
        "Add selection section" button is clicked, and passed
        a class_sched.ClassSchedule instance), or None if no
      callback is desired) ->
      swapArea := swapArea with a new SchedPanel child,
      unregistered and empty
      return that child with that callback ]

.swapArea: [ as passed to constructor, read-only ]
callback: [ as passed to constructor, read-only ]
.select(semSched, deptSched):
    [ (semSched is a SemesterSchedule instance) and
      (deptSched is a DeptSchedule instance) ->
      self := self displaying all the sections in
      (deptSched) under the heading of semSched.semName ]

State/Invariants:
._listboxFont: [ the tkFont.Font to use for self._sectBox]
._sectList:
    [ a list of class_sched.SectionSchedule instances
      corresponding to the lines shown in self._sectBox ]

Widgets and control variables:
._titleLabel: [ ttk.Label for the panel title ]
._titleVar: [ tk.StringVar for self._titleLabel ]
.addSectButton:
    [ a ttk.Button that adds the selected section to the
      current instructor's section list ]
.closeButton:
    [ a ttk.Button that calls self.close() and returns what
      it returns ]
._sectBox:
    [ a ScrolledList that displays section data for each
      section in self._sectList ]

Grid plan for self:
   0 1
+-----------------+---------------+
```

 cmsadds: A GUI for CMS course creation  
 New Mexico Tech Computer Center
24.1. SchedPanel.__init__()

```python
def __init__(self, swapArea, callback=None):
    """Constructor."""
    #-- 1
    # [ swapArea += a new ttk.Frame
    #   self := that ttk.Frame ]
    Transient.__init__(self, swapArea)
    self._listboxFont = tkFont.Font(family=NARROW_FONT_FAMILY, size=10)

    #-- 2
    self.swapArea = swapArea
    self.callback = callback

    #-- 3
    # [ self := self with all widgets and control variables
    #   created, and self._sectBox empty ]
    self._createWidgets()
```

24.2. SchedPanel._createWidgets()

```python
def _createWidgets(self):
    """Create all widgets and control variables."""
    rowx = colx = 0
    self._titleVar = tk.StringVar()
    self._titleVar.set('')
    self._titleLabel = ttk.Label(self,
                                 textvariable=self._titleVar,
                                 style=PANEL_TITLE_STYLE)
    self._titleLabel.grid(row=rowx, column=colx,
                          columnspan=2, sticky=tk.W)

    rowx, colx = rowx+1, 0
```
self._addSectButton = ttk.Button(self,
    text='Add selected section',
    command=self._addHandler)
self._addSectButton.grid(row=rowx, column=colx)

colx += 1
self._closeButton = ttk.Button(self,
    text='Close',
    command=self.close)
self._closeButton.grid(row=rowx, column=colx)

rowx, colx = rowx+1, 0
self._sectBox = ScrolledList(self, horiz=True,
    listboxArgs=dict(font=self._listboxFont,
                     width=self.LINE_WIDE, height=self.LINE_COUNT))
self._sectBox.grid(row=rowx, column=colx,
                   columnspan=2, padx=5, pady=5)

24.3. SchedPanel._addHandler(): Handler for Add selected section

def _addHandler(self):
    '''Handler for 'Add selected section'''
    
    If self.callback is None, no callback is registered; do nothing. If no line in the listbox is selected, display an error popup. Otherwise, call self.callback with the element of self._sectList corresponding to the selected line.

    #-- 1
    if self.callback is None:
        return

    #-- 2
    # [ if a line of self._sectBox is selected ->
    #   linex := index of that line as an int
    # else ->
    #   (screen) +=: an error popup
    # return ]
    linesTuple = self._sectBox.list.curselection()
    if len(linesTuple) == 1:
        rawLinex, = linesTuple
        linex = int(rawLinex)
    else:
        tkMessageBox.showerror("Error",
                               "No section is currently selected")
        return

    #-- 3
    self.callback(self._sectList[linex])
24.4. `SchedPanel.select()`: Change the displayed schedule

The listbox is cleared and then filled with lines for the sections in the given semester and department.

```python
#- - -  S c h e d  P a n e l . s e l e c t
def select(self, semSched, deptSched):
    """Change the displayed set of sections.
    ""
    #-- 1
    # [ self. sectBox := self. sectList, empty ]
    self._sectBox.list.delete(0, tk.END)

Next we build in self._sectList a list of the department's sections for this semester. Any error condition sets it to an empty list.

```python
#-- 2
# [ self._sectList +:= the class_sched.SectionSchedule
#     instances from deptSched ]
self._sectList = [sect
    for course in deptSched.genCourses() for sect in course.genSections() ]

Now set the text for the panel title to show the department code and name, and also repeat the current year and semester to emphasize to the user that this section list is restricted to that semester.

```python
#-- 3
# [ self. titleVar := deptCode, department name for deptCode
#     in classSched and semester name semSched.semName ]
text = ("{dn}\n{sem} section count {count}".format(
    sem=semSched.semName,
    dn=deptSched.deptName, count=len(self._sectList)))
self._titleVar.set(text)

Finally, fill the listbox with displays of the section's data; see Section 24.5, “SchedPanel._addSect(): Add one section line to the listbox” (p. 141).

```python
#-- 4
# [ self._sectBox += lines displaying the elements of
#     self._sectList in the same order ]
for sectSched in self._sectList:
    self._addSect(sectSched)
```

24.5. `SchedPanel._addSect()`: Add one section line to the listbox

```python
#- - -  S c h e d  P a n e l . _ a d d S e c t
def _addSect(self, sectSched):
    """Add one line to the section list.

```
The displayed line has four parts: the CRN; the full course number (e.g., "CH E 244L-06"), the course title, and a list of the instructor names in parentheses. However, display the parentheses only if there is at least one instructor.

```python
#-- 1
# [ if sectSched has any instructors ->
#   teachText := (slash-separated list of those
#   instructors' names)
# else ->
#   teachText := "?”
#]
nameList = [ name
   for name in sectSched.genInstructors() ]
if len(nameList) > 0:
   teachText = '/'.join(nameList)
else:
   teachText = '?”

#-- 2
text = ("{crn} {dc} {cn}{st}{sn} ({teach}) {title}".format(
   crn=sectSched.crn,
   dc=sectSched.deptCode,
   cn=sectSched.courseNo,
   st=sectSched.sectType,
   sn=sectSched.sectNo,
   title=sectSched.title,
   teach=teachText))
self._sectBox.list.insert(tk.END, text)
```

25. class SectionPanel: Edit section information

This class is a compound widget for editing information about one requested section: its title, the reload request if present, and the set of role assignments (other than the primary instructor role). It is a transient panel that appears in the swap area, so it inherits from Section 23, “class Transient() : Base class for panels in the swap area” (p. 136).

An important complication in the design of this panel is that it can be in an invalid state; see Section 6.1, “Control flow” (p. 22) for a general discussion of the implications: a call to the .close() method is a request, not a command, and can fail if the user is presented with a popup menu asking if they want to discard changes, and they say no. Currently, the only validity check is that the section title must be nonempty.

This class includes an instance of the people-finder widget described in Section 32, “Module uidfinder.py” (p. 176). Because it can take a perceptible amount of time to populate this widget with the set of user accounts from the TCC LDAP server, the constructor allows you to pass in a previously created UidFinder instance so the new instance can reuse the set of accounts from it.
class SectionPanel(Transient):
    '''Section request edit panel.

Exports:
    SectionPanel(app, swapArea, finder=None):
        [ (app is the containing App instance) and
          (swapArea is a ttk.Frame) and
          (finder is a uidfinder.UidFinder instance or None) ->
            swapArea := swapArea with a new SectionPanel child,
            unregistered, using uidFinder to supply the current
            set of TCC accounts if given ]
        return that new SectionPanel ]
    .app: [ as passed to constructor, read-only ]
    .swapArea: [ as passed to constructor, read-only ]
    .select(section):
        [ section is a readadds.SectionSchedule instance ->
          self := self editing (section) ]
    .close():
        [ if (the contents of self are valid) or
          (the user selects 'discard changes' in a popup) ->
            self.swapArea := self.swapArea with self removed
          else -> raise CloseFailure ]

State/Invariants:
    ._section:
        [ if select has never been called -> None
          else -> the current readadds.SectionSchedule instance ]
    ._titleFont:
        [ large font for panel title ]
    ._listboxFont:
        [ font for listbox of roleholders ]

Widgets and control variables:
    ._titleGroup: [ ttk.Frame ]
    ._titleLabel: [ ttk.Label for the panel title ]
    ._titleVar: [ tk.StringVar for self._titleLabel ]
    ._closeButton: [ ttk.Button to close the panel ]
    ._descGroup: [ ttk.Frame ]
    ._descEntry:
        [ ttk.Entry for editing self._section.desc ]
    ._descVar: [ tk.StringVar for self._descEntry ]
    ._descLabel: [ ttk.Label for self._descEntry ]
    ._reloadGroup: [ ttk.Frame ]
    ._reloadEntry:
        [ ttk.Entry to edit self._section.reload ]
    ._reloadVar: [ tk.StringVar for self._reloadEntry ]
    ._reloadLabel: [ ttk.Label for self._reloadEntry ]
    ._rolesFrame: [ ttk.Frame ]
    ._rolesGroup: [ ttk.LabelFrame ]
    ._rolesBox:
        [ a ScrolledList that shows non-advising-center role
          children of self._section ]
    ._delSelRoleButton:
        [ ttk.Button that deletes the selected role assignment ]
from self._rolesBox 
  ._advisingCBVar:
    [ tk.BooleanVar for self._advisingCB ]
  ._advisingCB:
    [ ttk.Checkbutton to control the readadds.ROLE_ADVISING role ]
  ._addRoleGroup: [ ttk.LabelFrame ]
  ._addRoleButtons:
    [ a list of ttk.Button instances corresponding to the elements of ROLE_BUTTON_LIST ]
  ._uidFinder:
    [ a uidfinder.UidFinder to specify a roleholder ]

Grid plan for self: 
+----------------+
0 | ._titleGroup |
+----------------+
1 | ._descGroup |
+----------------+
2 | ._reloadGroup |
+----------------+
3 | ._rolesFrame |
+----------------+
4 | ._addRoleGroup |
+----------------+
5 | ._uidFinder |
+----------------+

Grid plan for ._descGroup:
+-------------------------+
0 | ._descLabel | ._descEntry |
+-------------------------+

Grid plan for ._rolesFrame:
+-----------------------------+
0 | ._rolesGroup | ._addRoleGroup |
+-----------------------------+

Grid plan for ._reloadGroup:
+-----------------------------+
0 | ._reloadLabel | ._reloadEntry |
+-----------------------------+

Grid plan for ._rolesGroup:
+-----------------------------+
0 | ._rolesBox |
+-----------------------------+
1 | ._delSelRoleButton |
+-----------------------------+
2 | ._advisingCB |
+-----------------------------+
These class variables specify various widget attributes.

```
ROLE_BOX_WIDE = 34  # Width of roles listbox
ROLE_BOX_HIGH = 5   # Number of roles displayed
```

## 25.1. `SectionPanel.__init__()`

For the parent class, see Section 23, “class Transient(): Base class for panels in the swap area” (p. 136). For widget creation, see Section 25.2, “SectionPanel._createWidgets()” (p. 145).

```
# - - - S e c t i o n P a n e l . _ _ i n i t _ _

def __init__(self, app, swapArea, finder=None):
    '''Constructor.'
    ...
    #-- 1
    # [ swapArea := swapArea with a new Transient child
    #  self := that Transient ]
    Transient.__init__(self, swapArea)

    #-- 2
    self.app = app
    self.swapArea = swapArea
    self.finder = finder
    self._section = None
    self._titleFont = tkFont.Font(family=BASE_FONT_FAMILY,
                                  size=18)
    self._listboxFont = tkFont.Font(family=MONOSPACE_FAMILY,
                                     size=9)

    #-- 3
    # [ self := self with all widgets and control variables
    #  created ]
    self._createWidgets()
```

## 25.2. `SectionPanel._createWidgets()`

Widget creation is divided up according to the main areas:

- Section 25.3, “SectionPanel._createTitle(): Create panel title widgets” (p. 146).
- Section 25.5, “SectionPanel._createDesc(): Create description editing widgets” (p. 148).
- Section 25.7, “SectionPanel._createReload(): Create reload editing widgets” (p. 150).
- Section 25.10, “SectionPanel._createRolesGroup(): Create the role requests list” (p. 152).
- Section 25.13, “SectionPanel._createAddGroup(): Create the role add buttons” (p. 155).

```
# - - - S e c t i o n P a n e l . _ c r e a t e W i d g e t s
def _createWidgets(self):
```
Create all widgets and control variables.

#-- 1
# [ self := self with ._titleGroup and its children added
# and registered ]
rowx = 0
self._titleGroup = self._createTitle()
sel._titleGroup.grid(row=rowx, sticky=tk.W, padx=5, pady=5)

#-- 2
# [ self := self with ._descGroup and its children added
# and registered ]
rowx += 1
self._descGroup = self._createDesc()
sel._descGroup.grid(row=rowx, sticky=tk.W, padx=5, pady=5)

#-- 3
# [ self := self with ._reloadGroup and its children added
# and registered ]
rowx += 1
self._reloadGroup = self._createReload()
sel._reloadGroup.grid(row=rowx, sticky=tk.W, padx=5, pady=5)

#-- 4
# [ self := self with ._rolesFrame and its children added
# and registered ]
rowx += 1
self._rolesFrame = self._createRolesFrame()
sel._rolesFrame.grid(row=rowx, sticky=tk.W, padx=5, pady=5)

#-- 5
# [ if self.finder is None ->
#    self := self with ._uidFinder added, taking its
#    account set from TCC LDAP
# else ->
#    self := self with ._uidFinder added, taking its
#    account set from self.finder.accountList ]
rowx += 1
acctArg = (None
    if self.finder is None
    else self.finder.accountList)
sel._uidFinder = uidfinder.UidFinder(self,
    SEL_ROLEHOLDER_FINDER,
    accountList=acctArg)
sel._uidFinder.grid(row=rowx, sticky=tk.W, padx=5, pady=5)

25.3. SectionPanel._createTitle(): Create panel title widgets

```python
#-- SectionPanel._createTitle

def _createTitle(self):
    '''Creates the title and close button.''
```
See Section 25.4, “SectionPanel._closeHandler: Handler for the Close button” (p. 147).

```python
--- 3
# [ f := f with a new ttk.Button added that requests self's
#     panel to close
#     self._closeButton := that ttk.Button ]
colx += 1
self._closeButton = ttk.Button(f,
   text=CLOSE_BUTTON,
   command=self._closeHandler)
self._closeButton.grid(row=rowx, column=colx, padx=5, pady=5)

--- 4
return f

25.4. SectionPanel._closeHandler: Handler for the Close button

```
```python
#-- 2
# [ reply := a MultiDialog instance representing the user's
#     reply to choices POP_DISCARD_SECT and
#     POP_RETURN_TO_SECT ]
reply = MultiDialog(self, POP_SECT_TITLE, str(x),
    [POP_DISCARD_SECT, POP_RETURN_TO_SECT])

#-- 3
# [ if reply.result == POP_RETURN_TO_SECT ->
#     return
#   else ->
#     self._section := None
#     self.swapArea := self.swapArea with self removed ]
if reply.result == POP_RETURN_TO_SECT:
    return
else:
    self._section = None
    self.swapArea.clear()
```

### 25.5. `SectionPanel._createDesc()`: Create description editing widgets

```python
def _createDesc(self):
    '''Create the description editing widgets.
    '''
    #-- 1
    # [ self := self with a new ttk.Frame added, unregistered ]
    # f := that ttk.Frame ]
    f = ttk.Frame(self)

    #-- 2
    # [ f := f with a new ttk.Label added
    #   self._descLabel := that ttk.Label ]
    rowx, colx = 0, 0
    self._descLabel = ttk.Label(f,
        text=TITLE_LABEL)
    self._descLabel.grid(row=rowx, column=colx, padx=5, pady=5)

    See Section 25.6, “SectionPanel._descValidate(): Validate the section title” (p. 149).
```

```python
#-- 3
# [ self._descVar := a new tk.StringVar
# f := f with a new ttk.Entry added using self._descVar
# as its textvariable, that validates and stores
# in self._section.desc if valid, or complains
# if it is blank
# self._descEntry := that new ttk.Entry ]
        colx += 1
        self._descVar = tk.StringVar()
        self._descEntry = ttk.Entry(f,
            justify=tk.LEFT,
```

```python
New Mexico Tech Computer Center
```
25.6. SectionPanel._descValidate(): Validate the section title

This is the event handler called when the user leaves the section title entry. If it has any nonblank characters in it, it's considered valid, and the contents (after blank-stripping) are stored in the readadds.Section instance. If invalid, the user gets a popup error message and is not allowed to leave the widget.

Note the precondition that self._section has to have something in it. So long as the panel appears only when it is displaying a section, that is, only when .select() has been called, that precondition is met.

```python
def _descValidate(self, event):
    # Handle focus-out in the section description field.
    if self._section is a readadds.Section instance ->
        if self._descVar contains nonblank characters ->
            self._section.desc := contents of self._descVar
            with leading and trailing whitespace removed
        else -> (display) := an error popup
    ...
    # desc := current contents of self._descVar, with
    # leading and trailing spaces removed
    desc = self._descVar.get().strip()
```

Assuming that the description is okay, we must also inform the ReqPanel that it must rebuild its list of sections so that the updated description appears in that panel. See Section 14.45, “App.update ReqPanel(): Handler for a change in a metacourse name” (p. 88).

```python
if desc == '':
    title = "Invalid section title"
    text = ("You must enter a descriptive title for the "
            "section.")
```

25.7. SectionPanel._createReload(): Create reload editing widgets

```python
def _createReload(self):
    '''Create self._reloadGroup widgets.
    ```
    #-- 1
    # [ self := self with a new ttk.Frame added, unregistered
    #   f := that ttk.Frame ]
    f = ttk.Frame(self)

    #-- 2
    # [ f := f with a new ttk.Label added
    #   self._reloadLabel := that ttk.Label ]
    rowx, colx = 0, 0
    self._reloadLabel = ttk.Label(f,
       text=RELOAD_LABEL)
    self._reloadLabel.grid(row=rowx, column=colx, padx=5, pady=5)

    #-- 3
    # [ self._reloadVar := a new tk.StringVar
    #   f := f with a new ttk.Entry added using that tk.StringVar
    #   that updates self._section.reload
    #   self._reloadEntry := that ttk.Entry ]
    colx += 1
    self._reloadVar = tk.StringVar()
    self._reloadEntry = ttk.Entry(f,
       width=40,
       textvariable=self._reloadVar)
    self._reloadEntry.grid(row=rowx, column=colx, padx=5, pady=5)
    self._reloadEntry.bind("<FocusOut>",
       self._reloadValidate)

    #-- 4
    return f
```

See Section 25.8, “SectionPanel._reloadValidate(): Update the reload attribute” (p. 150).

25.8. SectionPanel._reloadValidate(): Update the reload attribute

When focus leaves the Reload from field, this handler is called to store the new value in the readadds.Section instance. Note the precondition that self._section must not be None; see the discussion in Section 25.6, “SectionPanel._descValidate(): Validate the section title” (p. 149) on how we argue that the precondition is satisfied.
# - - - SectionPanel._reloadValidate

def _reloadValidate(self, event):
    '''Handler for a change to the reload field.

    [ self._section is not None ->
      self._section.reload := contents of self._reloadVar,
      with leading and trailing whitespace removed ]
    '''

    #-- 1
    self._section.reload = self._reloadVar.get().strip()

25.9. SectionPanel._createRolesFrame()

Creates a frame to hold ._rolesGroup and ._addRoleGroup side by side. See Section 25.10, “SectionPanel._createRolesGroup(): Create the role requests list” (p. 152) and Section 25.13, “SectionPanel._createAddGroup(): Create the role add buttons” (p. 155).

# - - - SectionPanel._createRolesFrame

def _createRolesFrame(self):
    '''Create a frame around ._rolesGroup and ._addRoleGroup.

    #-- 1
    # [ self := self with a new ttk.Frame added
    #   f := that new ttk.Frame ]
    f = ttk.Frame(self)

    #-- 2
    # [ f := f with a new ttk.LabelFrame added containing
    #   self._rolesGroup and its children
    #   self._rolesGroup := that ttk.LabelFrame ]
    rowx, colx = 0, 0
    self._rolesGroup = self._createRolesGroup(f)
    self._rolesGroup.grid(row=rowx, column=colx,
                           padx=5, pady=5)

    #-- 3
    # [ f := f with a new ttk.LabelFrame added containing
    #   self._addRoleGroup and its children
    #   self._addRoleGroup := that ttk.LabelFrame ]
    colx += 1
    self._addRoleGroup = self._createAddGroup(f)
    self._addRoleGroup.grid(row=rowx, column=colx,
                             padx=5, pady=5)

    #-- 4
    return f
25.10. SectionPanel._createRolesGroup(): Create the role requests list

def _createRolesGroup(self, parent):  
    '''Create self._rolesGroup and its children.  
    '''  
    #-- 1  
    # [ parent := parent with a new ttk.LabelFrame added  
    # f := that ttk.Frame ]  
    f = ttk.LabelFrame(parent, text=ROLES_FRAME)  
    #-- 2  
    # [ f := f with a new ScrolledList added  
    # self._rolesBox := that ScrolledList ]  
    rowx = 0  
    self._rolesBox = ScrolledList(f,  
        listBoxArgs=dict(width=self.ROLE_BOX_WIDE,  
                        height=self.ROLE_BOX_HIGH,  
                        font=self._listboxFont))  
    self._rolesBox.grid(row=rowx, padx=5, pady=5)  

See Section 25.11, “SectionPanel._delSelRoleHandler: Handler for the Delete selected role button” (p. 153).

    #-- 3  
    # [ f := f with a new ttk.Button added that deletes the  
    # role corresponding to the selected element of  
    # self._rolesBox, if any  
    # self._delSelRoleButton := that ttk.Button ]  
    rowx += 1  
    self._delSelRoleButton = ttk.Button(f,  
        text=DEL_SEL_ROLE_BUTTON,  
        command=self._delSelRoleHandler)  
    self._delSelRoleButton.grid(row=rowx, padx=5, pady=5)  

See Section 25.12, “SectionPanel._advisingCBHandler: Handler for the Advising Center access checkbutton” (p. 154).

    #-- 4  
    # [ self._advisingCBVar := a new tk.BooleanVar  
    # f := f with a new ttk.Checkbutton added that adds  
    # or replaces the Advising Office role in  
    # self._section, using self._advisingCBVar as  
    # its control variable  
    # self._advisingCB := that ttk.Checkbutton ]  
    rowx += 1  
    self._advisingCBVar = tk.BooleanVar()  
    self._advisingCB = ttk.Checkbutton(f,  
        text=ADVISING_CHECKBUTTON,  
        variable=self._advisingCBVar,
SectionPanel._delSelRoleHandler: Handler for the Delete selected role button

This is the handler for the Delete selected role button. Note the preconditions: self._section must contain a readadds.Section; and the on-screen list of roleholders must be consistent with the readadds.Role children of self._section. We can assume that the roleholder list is up to date, but this method must also update it if a role is actually deleted.

```python
def _delSelRoleHandler(self):
    '''Handler for DEL_SEL_ROLE_BUTTON.

    [ (self._section is a readadds.Section instance) and
      (the lines in self._rolesBox corresponding to the
       children of self._section) ->
      if self._rolesBox has a selected line ->
        self._section := self._section without the
        readadds.Role instance corresponding to that line
        self._rolesBox := self._rolesBox without that line
      else ->
        (display) := error popup, no roleholder selected ]

    ...'
    #-- 1
    # [ if self._rolesBox has a selected line ->
    #  role := the readadds.Role instance corresponding to
    #  that line
    # else ->
    #  (display) := error popup, no roleholder selected
    #  return ]
    selectTuple = self._rolesBox.list.curselection()
    if len(selectTuple) == 0:
        title = "No roleholder selected"
        text = "Click on a roleholder to select it."
        tkMessageBox.showerror(title, text)
        return
    else:
        role = self._section[int(selectTuple[0])]
```

Note just above that when we index a readadds.Section instance, we use the indexing behavior inherited from et.Element, which selects a child element by its position. In the next step, we use the inherited et.Element.remove() method to remove the specified child. Updating of the list is handled by Section 25.17, “SectionPanel._rolesUpdate(): Rebuild the displays of roles” (p. 159).
#-- 2
# [ role is a child of self._section ->
#   self._section := self._section with role removed ]
self._section.remove(role)

#-- 3
# [ self._rolesBox := self._rolesBox rebuilt from
#   the children of self._section
#   self._advisingCB := shows presence or absence of
#   readadds.Role children of self._section with
#   kind=readadds.ROLE_ADVISING ]
self._rolesUpdate()

25.12. SectionPanel._advisingCBHandler: Handler for the Advising Center access checkbutton

This method is called when the self._advisingCB checkbutton changes state. It updates the set of role children of self._section to include or not include one for the advising center, depending on whether the checkbutton is set.

```python
# --- SectionPanel._advisingCBHandler

def _advisingCBHandler(self):
   '''Handle changes to the Advising Center role checkbox.

   [ self._section is a readadds.Section instance ->
     if (self._advisingCB is set) and
     (self._section has no child with .kind==
      readadds.ROLE_ADVISING) ->
     self._section := self._section with a new
      readadds.Role child with .kind=
      readadds.ROLE_ADVISING and .email=ADVISING_EMAIL
     else if (self._advisingCB is clear) and
     (self._section has a child with .kind==
      readadds.ROLE_ADVISING) ->
     self._section := self._section without that child
     else -> I ]
   ...
   #-- 1
   # [ if self._section has a readadds.Role child with its
   #   .kind==readadds.ROLE_ADVISING ->
   #   role := that readadds.Role
   #   else ->
   #   role := None ]
   role = None
   for child in self._section:
     if child.kind == readadds.ROLE_ADVISING:
       role = child

   The call to the Section.remove() method, which deletes the role when the checkbutton is turned off, is inherited from et.Element.
```
25.13. SectionPanel._createAddGroup(): Create the role add buttons

This method creates self._addRoleGroup and its contents. The list self._addRoleButtons is a sequence of button widgets corresponding to the elements of Section 9.35.37, “ROLE_BUTTON_LIST” (p. 35).

```python
# - - - SectionPanel._createAddGroup

def _createAddGroup(self, parent):
    '''Set up self._addRoleGroup and its children.
    '''
    #-- 1
    self._addRoleButtons = []
    #-- 2
    # [ parent := parent with a new ttk.LabelFrame added, #
    #   unregistered
    #   f := that ttk.LabelFrame ]
    f = ttk.LabelFrame(parent, text=ADD_SEL_ROLE_FRAME)
    #-- 3
    # [ f := f with a sequence of ttk.Button widgets #
    #   added corresponding to the role kinds and role #
    #   names from ROLE_BUTTON_LIST, displaying the role #
    #   name, with a command that adds the currently selected #
    #   user (if any) from self._uidFinder with that role to #
    #   self._section
    #   self._addRoleButtons +=: those ttk.Button widgets ]
    rowx = 0
    for roleKind, roleName in ROLE_BUTTON_LIST:
        #-- 2 body
        # [ f := f with a new ttk.Button added displaying
```
# (roleName), with a command that adds the currently
# selected user (if any) from self._uidFinder with
# role (roleKind) to self._section
# self._addRoleButtons += that ttk.Button
button = self._makeRoleButton(f, roleKind, roleName)
self._addRoleButtons.append(button)
button.grid(row=rowx, padx=5, pady=5, sticky=tk.W)
rowx += 1

#-- 4
return f

25.14. SectionPanel._makeRoleButton(): Create one add-role button

def _makeRoleButton(self, f, roleKind, roleName):
    '''Add one button that creates a role request.
    [ (self._section is a readadds.Section instance) and
      (f is a ttk.Frame) and
      (roleKind is a role code) and
      (roleName is a role name) ->
      f := f with a new ttk.Button added displaying
        (roleName), with a command that adds the currently
        selected user (if any) from self._uidFinder with
        role (roleKind) to self._section
      ]
    return that ttk.Button ]
    ...
    #-- 1
    # [ handler := a function that adds the currently selected
    #    user (if any) from self._uidFinder with role (roleKind)
    #    to self._section ]
    def handler():
        '''Closure to add a given role of kind (roleKind)
        ...
        self._roleHandler(roleKind)
        ...
    #-- 2
    # [ f := f with a new ttk.Button added that calls handler
    #   button := that ttk.Button ]
    button = ttk.Button(f,
                          command=handler,
                          text=roleName)
    #-- 3
    return button
25.15. **SectionPanel._roleHandler()**: Generic handler for an add-role button

```python
#--- SectionPanel._roleHandler

def _roleHandler(self, roleKind):
    '''Generic handler for the buttons that add role requests.

    [ (self._section is a readadds.Section instance) and
      (roleKind is a role code) ->
      if self._uidFinder has a selected entry ->
      self._section := self._section with a new
      readadds.Role child added using kind=roleKind
      and the uid from the selected entry of
      self._uidFinder
    else ->
      (display) +:= an error popup ]

    #-- 1
    # [ if self._uidFinder has a selected entry ->
    #   tccAccount := a uidfinder.TccAccount instance
    #     representing that entry
    # else ->
    #   (display) +:= an error popup
    # return ]
    tccAccount = self._uidFinder.get()
    if tccAccount is None:
        title = "No roleholder selected"
        text = ("To add a person in a given role, first "
        "select that person in the '{0}' area, "
        "then click the button in the '{1}' area "
        "corresponding to the desired role.".format(
        SEL_ROLEHOLDER_FINDER, ADD_SEL_ROLE_FRAME))
        tkMessageBox.showerror(title, text)
        return

    #-- 2
    # [ if tccAccount.uid == self._section.teacher().email ->
    #   (display) +:= error popup
    # return
    # else -> I ]
    primary = self._section.teacher()
    if primary.email == tccAccount.uid:
        title = "Duplicate role assignment"
        text = ("UID {0} is the primary instructor for this "
        "section.".format(primary.email))
        tkMessageBox.showerror(title, text)
        return
```

Now that we know who is being added in this role, check for duplicates. The person must not be the primary instructor, or in any other role.
Now it is safe to add the new Role child.

Finally we call Section 25.17, “SectionPanel._rolesUpdate(): Rebuild the displays of roles” (p. 159) to redraw the listbox of role requests so that it corresponds to the current set of children of self._section.

25.16. SectionPanel.select(): Display a readadds.Section instance

def select(self, section):
    '''Display a readadds.Section instance in self.''
    #-- 1
    # [ self.swapArea := self.swapArea displaying self
    #   self.section := section ]
    self.swapArea.select(self)
    self._section = section

    #-- 2
    # [ self._descVar := section.desc
    #   self._reloadVar := section.reload
    self._descVar.set(section.desc)

    #-- 3
    # [ if self._section has a readadds.Role child whose
    #   email == tccAccount.uid ->
    #   (display) += error popup
    #   return
    # else -> I ]
    for role in self._section:
        if role.email == tccAccount.uid:
            title = "Duplicate role assignment"
            text = ("UID {0} already has role '{1}' in this "
                    "section.".format(role.email, role.kind))
            tkMessageBox.showerror(title, text)
            return

    #-- 4
    # [ self._section += a new readadds.Role instance
    #   with kind=roleKind and email=tccAccount.uid ]
    role = readadds.Role(self._section, roleKind, tccAccount.uid)

    #-- 5
    # [ self._rolesBox := self._rolesBox rebuilt from
    #   the children of self._section
    #   self._advisingCB := shows presence or absence of
    #   readadds.Role children of self._section with
    #   kind=readadds.ROLE_ADVISING ]
    self._rolesUpdate()
See Section 25.17, “SectionPanel._rolesUpdate(): Rebuild the displays of roles” (p. 159).

25.17. SectionPanel._rolesUpdate(): Rebuild the displays of roles

First we clear self._rolesBox and self._advisingCB. Then we iterate over the role children of self._section. If that child has a kind attribute of readadds.ROLE_ADVISING, set the advising checkbox. Otherwise, add a line to self._rolesBox displaying the kind, email, and full name of the roleholder.

As we iterate over the role children of the section, we look up their user IDs in the self._uidFinder. This could conceivably fail, in which case we silently delete the record. See Section 25.18, “SectionPanel.showRole(): Display one role request” (p. 160).

There is potential for a subtle defect here: what happens if the self._showRole() method deletes a child of self._section in the middle of this loop, which iterates over the children of self._section? To forestall this problem, we make a copy of the list of children and use that to iterate at this level. Then self._showRole() can remove roles for nonexistent accounts under the actual readadds.Section instance, which maintains the invariant that the set of role children corresponds to the displayed lines in self._roleBox.
# child with kind=readadds.ROLE_ADVISING
# self._rolesBox.list +=: lines displaying the kind,  
# email, and TCC full name from self._uidFinder  
# for children of other kinds, provided that  
# self._uidFinder
roleList = [role  
for role in self._section]
for role in roleList:
    #-- 2 body
    # [ role is a readadds.Role instance ->  
    # if role.kind == readadds.ROLE_ADVISING ->  
    # self.advisingCBVar := True  
    # else if role.email is not in self._uidFinder ->  
    # (display) +=: error popup  
    # self._section := self._section with role deleted  
    # else ->  
    # self._rolesBox.list +=: a line displaying  
    # role.kind, role.email, and the TCC full name  
    # from self._uidFinder for role.email ]
self._showRole(role)

25.18. SectionPanel.showRole(): Display one role request

This method is used to rebuild the components of self.rolesGroup initially or after a change in the set of role requests.

Requests for the Advising Center role are handled separately. The uid is always the same: Section 9.9, “ADVISING_EMAIL” (p. 26). Hence we use a checkbutton for this option, and it does not correspond to a line in self._rolesBox.

```python
# - - - S e c t i o n P a n e l . _ s h o w R o l e

def _showRole(self, role):
    '''Display one role request.

    [ (self._section is a readadds.Section instance) and
    (role is a readadds.Role child of self._section) ->
    if role.kind == readadds.ROLE_ADVISING ->
    self.advisingCBVar := True
    else if role.email is not in self._uidFinder ->
    (display) +=: error popup
    self._section := self._section with role deleted
    else ->
    self._rolesBox.list +=: a line displaying
    role.kind, role.email, and the TCC full name
    from self._uidFinder for role.email ]
    ...

    #-- 1
    if role.kind == readadds.ROLE_ADVISING:
    self.advisingCBVar.set(True)
    return
```
In the `adds-season-yyyy.xml` file, roles are identified only by their TCC user ID (uid). So that the user can be sure of the actual identity of the roleholder, we use the `UidFinder.lookupUid()` method to translate this to a `uidfinder.TccAccount` instance. This can fail: an existing `adds-season-yyyy.xml` file may contain an entry for an account that has since been deleted from LDAP. In that case we must also delete the Role child of `self._section`.

```python
try:
tccAccount = self._uidFinder.lookupUid(role.email)
except KeyError:
title = "Unknown TCC user ID"
text = ("Account '{0}' is not defined in the current "
         "LDAP; deleting role '{1}' for {2}".format( role.email, role.kind, self._section.fullCode))
tkMessageBox.showerror(title, text)
self._section.remove(role)
return
```

Now build the line to be displayed in `self._rolesBox.list`.

```python
#-- 3
# [ self._rolesBox := self._rolesBox with a new line added
#   at the end, displaying role.kind, role.email, and
#   tccAccount.fullName ]
text = ("{0} {1:<8} {2}".format( role.kind, role.email, tccAccount.fullName))
self._rolesBox.list.insert(tk.END, text)
```

### 25.19. `SectionPanel.close()`

See Section 25, “class `SectionPanel`: Edit section information” (p. 142) for a discussion of invalid states for this panel and the consequences.

```python
#-- - SectionPanel.close

def close(self):
    '''Handle a close request.

    [ if the contents of self are valid ->
      self.swapArea := self.swapArea with self removed
    else -> raise CloseFailure ]
    '''
    #-- 1
desc = self._descVar.get().strip()
if desc != '':
```
class MetaPanel: Edit cross-listed sets (metacourses)

This class is a compound widget that edits the set of metacourses for a given primary instructor. It is a transient panel that inherits from Section 23, “class Transient(): Base class for panels in the swap area” (p. 136).

Once this widget is displaying a readadds.CrossParent instance, there are certain validity conditions that it must enforce:

• A metacourse must have a title. Any nonblank string will do for our purposes. At this writing (June 2013) metacourses are not built through IMS file imports anyway; the records in the adds-season-yyyy.xml file are just to record what needs to be done by hand.

• A metacourse must have at least two sections.

• All the sections in a metacourse must also be in the set of sections requested by the same instructor. This condition is enforced by the readadds module, which refuses to read an adds-season-yyyy.xml file that fails to meet this condition, and raises a readadds.IntegrityError if any method call would violate it.

Hence, the .close() method must check the first two conditions before closing the panel successfully. If either condition is not met, the user will get a popup asking if they want to discard the half-build metacourse. If they select Yes, we’ll need to delete the associated readadds.CrossParent from the model, but this requires cleanup operations in other places: App must blank the transient panel, and ReqPanel must remove the metacourse from its MetaList widget. All those operations are accomplished by calling Section 14.42, “App.deleteMeta(): Remove a metacourse” (p. 87).
[ if (the contents of self are valid) or
 (the user selects 'discard changes' in a popup) ->
  self.app := self.app with the swap area blanked
  and self._crossParent removed from its ReqPanel
  self._crossParent := None
else -> raise CloseFailure ]

State/Invariants:
._crossParent:
  [ if self is not editing a metacourse -> None
  else ->
    the readadds.CrossParent instance being edited ]

Widgets and control variables:
._metaPanelFrame: [ ttk.Frame ]
  ._metaPanelTitle: [ ttk.Label for the panel title ]
  ._closeButton: [ ttk.Button that closes self ]
._metaDescFrame: [ ttk.Frame ]
  ._metaDescLabel: [ ttk.Label for the metacourse description ]
  ._metaDescEntry: [ ttk.Entry for the metacourse description ]
  ._metaDescVar: [ tk.StringVar for ._metaDescEntry ]
  ._reloadLabel: [ ttk.Label for the course to reload ]
  ._reloadEntry: [ ttk.Entry for the course to reload ]
  ._reloadVar: [ tk.StringVar for ._reloadEntry ]
._controlFrame: [ ttk.Frame ]
  ._addSectMetaButton:
    [ ttk.Button that adds the selected element of the
      section list in self.app's ReqPanel, if there is one,
      to the child sections in self ]
  ._delSectMetaButton:
    [ ttk.Button that deletes the child element selected
      in self._childrenBox, if any ]

Each child section of a readadds.CrossParent is a readadds.Child instance, which contains only
the CRN of the child section. When the user adds a CRN, we ask the app to translate that into a
readadds.Section instance so we can display the course number and title using Section 19, “class
SectionsList: A scrollable listbox of section lines” (p. 119). The actual current set of
readadds.Section instances is kept inside that SectionsList. Note the invariant: the set of CRNs
in the readadds.Child elements under self._crossParent must be identical to the set of CRNs
of sections currently displayed in the ._childrenBox SectionsList widget.
Class variables: the width of the description and reload fields.

DESC_ENTRY_WIDTH = 40
RELOAD_ENTRY_WIDTH = 40

### 26.1. `MetaPanel.select()`: Display a metacourse

```python
def select(self, crossParent):
    '''Display a set of cross-listed courses.
    '''
    #-- 1
    self._crossParent = crossParent
    self._metaDescVar.set(crossParent.desc)
    if crossParent.reload is not None:
        self._reloadVar.set(crossParent.reload)

    #-- 2
    # [ self._childrenBox := self._childrenBox, cleared ]
    self._childrenBox.clear()

    #-- 3
    # [ self._childrenBox := self._childrenBox displaying
    #   the child sections of crossParent ]
```

All that `crossParent` knows is the CRNs of the child sections. We must translate these into `readadds.Section` instances by using Section 14.25, “`App.lookupCRN()`: Find a CRN in the current semester” (p. 75). We can be sure that every child section is a section under that instructor because it is an integrity constraint enforced by the `readadds` module.
26.2. MetaPanel.__init__()

For the parent class, see Section 23, “class Transient(): Base class for panels in the swap area” (p. 136). For widget creation, see Section 26.3, “MetaPanel.createWidgets()” (p. 165).

```python
# - - - M e t a P a n e l . __ i n i t _ _

def __init__(self, swapArea, app, updater):
    '''Constructor.
    '''
    #-- 1
    # [ swapArea := swapArea with a new Transient child
    #  self := that Transient ]
    Transient.__init__(self, swapArea)
    #-- 2
    self.swapArea = swapArea
    self.updater = updater
    self.app = app
    self._crossParent = None
    #-- 3
    # [ self := self with all widgets and control variables
    #  created ]
    self._createWidgets()
```

26.3. MetaPanel.createWidgets()

See Section 26.4, “MetaPanel._createTitle(): Create the panel title and close button” (p. 166).

```python
# - - - M e t a P a n e l . c r e a t e W i d g e t s

def _createWidgets(self):
    '''Create all widgets.
    '''
    #-- 1
    rowx = 0
```
26.4. MetaPanel._createTitle(): Create the panel title and close button

```python
def _createTitle(self):
    '''Create self._metaPanelFrame and its contents.
    '''
    #-- 1
    f = ttk.Frame(self)

    #-- 2
    rowx, colx = 0, 0
    self._metaPanelTitle = ttk.Label(f,
        style=PANEL_TITLE_STYLE,
        text=META_PANEL_TITLE)
    self._metaPanelTitle.grid(row=rowx, sticky=tk.W,
        padx=5, pady=5)

    #-- 3
    colx += 1
    self._closeButton = ttk.Button(f,
        command=self._closeHandler,
        text=CLOSE_BUTTON)
    self._closeButton.grid(row=rowx, column=colx,
        padx=5, pady=5)
```

See Section 26.6, “MetaPanel._createDescFrame(): Create title and reload fields” (p. 167).

```python
#-- 2
rowx += 1
self._metaDescFrame = self._createDescFrame()
self._metaDescFrame.grid(row=rowx, sticky=tk.W,
    padx=5, pady=5)

See Section 26.9, “MetaPanel._createControlFrame(): Create panel controls” (p. 169).

```python
#-- 3
rowx += 1
self._controlFrame = self._createControlFrame()
self._controlFrame.grid(row=rowx, sticky=tk.W,
    padx=5, pady=5)

#-- 4
rowx += 1
self._childrenBox = SectionsList(self)
self._childrenBox.grid(row=rowx, sticky=tk.W,
    padx=5, pady=5)
```
26.5. MetaPanel._closeHandler(): Handler for the Close button

```python
#-- 4
return f
```

26.6. MetaPanel._createDescFrame(): Create title and reload fields

```python
#-- 1
f = ttk.Frame(self)
#-- 2
rowx, colx = 0, 0
self._metaDescLabel = ttk.Label(f,
    text=META_DESC_LABEL)
self._metaDescLabel.grid(row=rowx, column=colx, sticky=tk.E,
                         padx=5, pady=5)

When focus leaves the title field, the title must be nonempty; this is checked by Section 26.7, “MetaPanel._descValidate(): Validate the metacourse description field” (p. 168), which also stores the stripped title in the readadds.CrossParent instance.

```
When focus leaves the reload field, the value in the `readadds.CrossParent` instance is updated by Section 26.8, “MetaPanel._updateReload(): Update the reload field” (p. 169).

```python
#-- 5
colx += 1
self._reloadVar = tk.StringVar()
self._reloadEntry = ttk.Entry(f,
    width=self.RELOAD_ENTRY_WIDTH,
    justify=tk.LEFT,
    textvariable=self._reloadVar)
self._reloadEntry.grid(row=rowx, column=colx,
    sticky=tk.W, padx=5, pady=5)
self._reloadEntry.bind("<FocusOut>", self._updateReload)
#-- 6
return f
```

26.7. MetaPanel._descValidate(): Validate the metacourse description field

When the user leaves the `_metaDescEntry` field, there must be a title in it. The user gets to ignore the popup if they wish; the lack of a title will be detected on the call to Section 26.13, “MetaPanel.close()” (p. 172). Assuming there is text in the field, we store it into the underlying `CrossParent.desc` attribute.

When the user changes the description, we must also call the `self.updater` callback so that the updated description will appear in the `ReqPanel`’s list of metacourses; see Section 14.45, “App.updateReqPanel(): Handler for a change in a metacourse name” (p. 88).

```python
#-- descValidate

def _descValidate(self, event):
    """Handle FocusOut from self._metaDescEntry."
    """
    #-- 1
    # [ text := contents of self._metaDescVar, stripped ]
    text = self._metaDescVar.get().rstrip()
    #-- 2
    # [ if text is empty ->
    # (display) +:= an error popup
    # return
    # else ->
    # self._crossParent.desc = text
    if len(text) == 0:
        title = "Invalid metacourse title"
```
Please enter a title for this metacourse.
tkMessageBox.showerror(title, text)
return
else:
    self._crossParent.desc = text

#-- 3
if self.updater is not None:
    self.updater()

26.8. MetaPanel._updateReload(): Update the reload field

def _updateReload(self, event):
    '''Handle FocusOut from self._reloadEntry.
    ...'''
    self._crossParent.reload = self._reloadVar.get().rstrip()

26.9. MetaPanel._createControlFrame(): Create panel controls

def _createControlFrame(self):
    '''Create a frame for buttons.
    ...'''
    #-- 1
    f = ttk.Frame(self)

    See Section 26.10, “MetaPanel._addSectHandler(): Handle the Add selected section button” (p. 170).

    #-- 2
    rowx, colx = 0, 0
    self._addSectMetaButton = ttk.Button(f,
                                         command=self._addSectHandler,
                                         text=ADD_SECT_META_BUTTON)
    self._addSectMetaButton.grid(row=rowx, column=colx,
                                  padx=5, pady=5)

    See Section 26.11, “MetaPanel._delSectHandler(): Handler for the Delete selected section button” (p. 171).

    #-- 3
    colx += 1
    self._delSectMetaButton = ttk.Button(f,
                                          command=self._delSectHandler,
                                          text=DEL_SECT_META_BUTTON)
    self._delSectMetaButton.grid(row=rowx, column=colx,
                                  padx=5, pady=5)
26.10. MetaPanel._addSectHandler(): Handle the Add selected section button

To add the section that is selected in the ReqPanel, we need to know which section is selected. Since we don't have direct access to the ReqPanel, we ask the application indirectly through Section 14.43, “App.getSelectedSection(): What section is selected in the ReqPanel?” (p. 87).

```
def _addSectHandler(self):
    """Handler for the Add selected section button."
    #-- 1
    # [ if app's ReqPanel has a selected section in its section list ->
    #   newSection := the corresponding readadds.Section instance
    # else ->
    try:
        newSection = self.app.getSelectedSection()
    except ValueError:
        title = "No section is selected"
        text = ("To add a section to this metacourse, first "
                "select it in the list of sections under the "
                "primary instructor, then click the "
                "Add selected section button."
        )
        tkMessageBox.showerror(title, text)
    return

    #-- 2
    # [ crnList := list of the CRNs of self._crossParent ]
    crnList = [section.crn
                for section in self._childrenBox.genSections()]

    #-- 3
    # [ if section.crn is in crnList ->
    #   (display) +:= an error popup
    #   return
    # else -> I ]
    if newSection.crn in crnList:
        title = "Duplicate child section"
        text = ("Section {s.crn} ({s.fullCode}) is already in "
                "this metacourse."
        )
        tkMessageBox.showerror(title, text)
    return
```

Although it wouldn't currently do any harm, we don't want the same child section in the metacourse more than once, so make sure it isn't already there.
Append the new section to `self._childrenBox`, and also add a new `readadds.Child` element to the model, to maintain the invariant that the set of sections displayed in the listbox is identical to the set of child sections in the model.

```python
#-- 4
# [ self._crossParent += a new readadds.Child made from section
# self._childrenBox := self._childrenBox with section added ]
child = readadds.Child(self._crossParent, newSection.crn)
self._childrenBox.add(newSection)
```

26.11. `MetaPanel._delSectHandler()`: Handler for the *Delete selected section* button

```python
#-- - - - M e t a P a n e l . _ d e l S e c t H a n d l e r

def _delSectHandler(self):
    '''Handler for the "Delete selected section" button.
    '''
    #-- 1
    # [ if a line in self._childrenBox is selected ->
    #   section := the corresponding readadds.Section
    # else ->
    #   (display) +=: an error popup
    #   return ]
    section = self._childrenBox.get()
    if section is None:
        title = "No section is selected"
        text = ("To delete a child section from this "
               "metacourse, select its line in the listbox "
               "and then click '{0}'.".format(Del_sect_meta_button))
        tkMessageBox.showerror(title, text)
        return
    #-- 2
    # [ self._childrenBox := self._childrenBox with
    #   section removed
    # self._crossParent := self._crossParent with the child
    # removed that has CRN=section.crn ]
    self._childrenBox.delete(section)
    self._crossParent.delete(section.crn)
```

We have to remove the child in two places: from the `self._childrenBox` widget, and from the underlying `readadds.CrossParent` instance.

26.12. `MetaPanel.clear()`: Clear out the content

This method erases all widgets and sets `self._crossParent` to `None`. 
# - - - MetaPanel.clear

def clear(self):  
    '''Clear all widgets and state.  
    ...  
    self._metaDescVar.set('')  
    self._reloadVar.set('')  
    self._childrenBox.clear()  
    self._crossParent = None  

26.13. MetaPanel.close()

There are two validity conditions for this panel.

- There must be a title (description) for the metacourse.
- There must be at least two child sections.

# - - - MetaPanel.close

def close(self):
    '''Try to close this panel.  
    ...  
    #-- 1  
    # [ if self._metaDescVar, stripped, is empty ->  
    #   (display) += an error popup  
    #   return  
    # else -> I ]  
    desc = self._metaDescVar.get().strip()
    if desc == '':
        reply = MultiDialog(self, POP_BAD_META_TITLE,
                             POP_BAD_META_TEXT,
                             [POP_BAD_META_DELETE, POP_BAD_META_CANCEL])
        if reply.result == POP_BAD_META_DELETE:
            self.app.deleteMeta(self._crossParent)
            return
        else:
            raise CloseFailure(POP_BAD_META_TEXT)

    #-- 2  
    # [ if self._childList has fewer than two members ->  
    #   (display) += an error popup  
    #   raise CloseFailure  
    # else -> I ]  
    if self._childrenBox.nSections() < 2:
        reply = MultiDialog(self, POP_MISSING_KIDS, POP_TWO_KIDS_REQ,
                             [POP_DISCARD_META, POP_BAD_META_CANCEL])
        if reply.result == POP_BAD_META_CANCEL:
            raise CloseFailure(POP_TWO_KIDS_REQ)
        self.app.deleteMeta(self._crossParent)
        return
To clear out all our widgets, we call Section 26.12, “MetaPanel.clear(): Clear out the content” (p. 171). Then we clear the swap area.

```python
#-- 3
# [ self := self with all widgets cleared
#  self := self removed from self.swapArea ]
self.clear()
self.swapArea.clear()
```

### 27. class MultiDialog: Dialog popup

The `tkMessageBox` module is useful for many simple popup dialog boxes, but each popup supports only two choices, and the legends on the buttons, such as `Yes` or `Cancel`, cannot be customized.

Fredrik Lundh, who wrote `Tkinter`, has written a standard Python module that can be used as a basis for customized popup. See the documentation for this `tkSimpleDialog` module.

Our `MultiDialog` class provides this functionality:

- A window title and a block of text.
- A vertically stacked row of two or more buttons. Vertical stacking allows us to have longer, more meaningful labels such as “Save adds-fall-2019.xml” or “Discard all your changes”.

To customize the base `tkSimpleDialog.Dialog` class, one must override certain methods:

- We define a `.body(master)` method that creates the widgets in the popup other than the various buttons, placing them in the widget specified by the `master` argument.

  In our case, Section 27.2, “MultiDialog.body(): Place widgets in the popup” (p. 174) just adds a `ttk.Label` displaying the `text` argument to the constructor.

- We define Section 27.3, “MultiDialog.buttonbox(): Create the choice buttons” (p. 174) method that creates the action buttons at the bottom of the popup.

Here is the interface.

```python
# - - - - - c l a s s  M u l t i D i a l o g
class MultiDialog(tkSimpleDialog.Dialog):
    '''Dialog with arbitrary number of buttons in a vertical row.

    Exports:
    MultiDialog(parent, title, text, buttonList):
        [ (parent is a tk.Frame) and
          (title is a str) and (text is a str) and
          (buttonList is a list of str values) ->
            return the result of a popup dialog with title=(title),
            text=(text), and a vertical row of buttons whose labels
            are the elements of buttonList, and the result is a
            Dialog instance whose .result attribute is the label from the
            button that was pressed ]
```

27.1. MultiDialog.__init__()

Our constructor accepts two arguments that are not required by the parent constructor. The text argument is the explanatory text on the popup, and buttonList is a list of strings that specify the buttons. We'll store those arguments in instance attributes where they will be picked up by the .body() and .buttonbox() methods, then call the parent constructor.

cmsadds
```python
# - - - M u l t i D i a l o g . __ i n i t _ _

def __init__(self, parent, title, text, buttonList):
    '''Constructor.'''
    
    #-- 1
    self.text = text
    self.buttonList = buttonList
    self._textFont = ('Liberation Serif', 16)

    #-- 2
    # [ self := a new tkSimpleDialog.Dialog with title=(title)
    #     and its background color set to MAIN_BG_COLOR ]
    tkSimpleDialog.Dialog.__init__(self, parent, title)
```

cmsadds

27.2. MultiDialog.body(): Place widgets in the popup

Aside from the buttons at the bottom, the only widget in the popup is the ttk.Label showing the explanatory text.

cmsadds
```python
# - - - M u l t i D i a l o g . b o d y

def body(self, master):
    '''Put up the body (the part above the buttons).'''
    
    #-- 1
    # [ master += a new ttk.Label containing self.text ]
    self._textLabel = tk.Label(master, text=self.text,
                               font=self._textFont)
    self._textLabel.grid()
    return master
```

cmsadds

27.3. MultiDialog.buttonbox(): Create the choice buttons

This method replaces the method in the base class that creates the various buttons. The number of buttons and their labels are given by the self.buttonList attribute. Each button has a handler that stores that button's text into the .result attribute of the MultiDialog instance returned to the caller, so they can determine which button was clicked.
Warning

The author's first attempts at this method ran into a very subtle and annoying bug. After creating a frame f to hold the buttons, the first version registered that frame using .grid(). However, the rest of the base class uses the .pack() geometry manager, and the two cannot be mixed within the same widget.

The application went into a hard loop, but not at the point where .grid() was called. The problem did not manifest until the base class constructor's call to .deiconify(), at which point the application hangs.

```python
# - - - M u l t i D i a l o g . b u t t o n b o x

def buttonbox(self):
    '''Display the buttons.'''
    #-- 1
    # [ self += a new tk.Frame
    #   f := that tk.Frame ]
    f = tk.Frame(self)
    f.pack() # NB: Don't use grid here! It hangs.

    for rowx, buttonText in enumerate(self.buttonList):
        def handle(k=rowx):
            self.result = self.buttonList[k]
            self.ok()
        b = tk.Button(f, text=buttonText, command=handle,
                      font=self._textFont)
        b.grid(padx=5, pady=5, sticky=tk.E+tk.W)
```

Each button gets its own handle() closure that sets self.result to its label and then calls tkSimpleDialog.ok() in the base class, which removes the popup.

```python
#-- 2
# [ f += a vertical stack of buttons corresponding to the
#   elements of self.buttonList such that each button
#   sets self.result to its text and removes the
#   popup ]
for rowx, buttonText in enumerate(self.buttonList):
    def handle(k=rowx):
        self.result = self.buttonList[k]
        self.ok()
    b = tk.Button(f, text=buttonText, command=handle,
                  font=self._textFont)
    b.grid(padx=5, pady=5, sticky=tk.E+tk.W)
```

28. addsFileName(): The name of the save file

This method determines the name under which the current request set is to be saved. See the Section 10.1, “adds-file-name: What is the name of the XML file?” (p. 38) specification function.

```python
# - - - a d d s F i l e N a m e

def addsFileName(semSched):
    '''Return the file name where the current request set lives.'''
    return ("{0}{1}-{2}{3}".format(
        ADDS_PREFIX,
        class_sched.SEM_CODE_TO_NAME[semSched.semCode].lower(),
```
29. **message()**: Write a message to stderr

```python
# --- message
def message(*L):
    '''Print a message and terminate.
    L is a list of strings ->
    sys.stderr += concatenated elements of L
    stop execution]
    ...
    print(''.join(L), file=sys.stderr)
```

30. **fatal()**: Write a message and terminate

```python
# --- fatal
def fatal(*L):
    '''Print a message and terminate.
    L is a list of strings ->
    sys.stderr += concatenated elements of L
    stop execution]
    ...
    message(*L)
    sys.exit(1)
```

31. **Epilogue**

```python
# --- Epilogue
if __name__ == '__main__':
    main()
```

32. **Module uidfinder.py**

This module defines a compound widget containing the UID finder described in Section 3.5, “Design of the UID finder” (p. 13). It resides in a separate file to facilitate potential reuse in other applications. The first line allows it to be run as a standalone script, in which case it uses the test driver in Section 32.23, “uidfinder.py: Test driver” (p. 194).

The application has two of these widgets (one for finding primary instructors and one for finding role-holders), but we would like to avoid building the account list from LDAP twice. Hence, the optional
accountList argument allows the second instance to use the .accountList argument from the first instance.

32.1. uidfinder: Prologue

```python
#!/usr/bin/env python
'''uidfinder.py: Tkinter compound widget for finding TCC accounts.

Exports:
UidFinder(parent, title, callback=None, accountList=None):
    [ (parent is a widget) and
        (title is the title to display in self's LabelFrame) and
        (callback is a function that will be called with a
        TccAccount instance as its only argument whenever the
        user clicks on a displayed entry, defaulting to
        no callback) and
        (accountList is a list of TccAccount instances, defaulting
to the list extracted from TCC LDAP) ->
            if accountList is None ->
                if TCC LDAP is available ->
                    parent := parent with a new, ungridded child that
                        allows search for TCC LDAP UIDs and personal names
                    return a new UidFinder widget with those arguments
                else -> raise IOError
            else ->
                parent := parent with a new, ungridded child that
                allows search for names from accountList
                return a new UidFinder widget with those arguments ]

    .title: [ as passed to the constructor ]
    .callback: [ as passed to the constructor ]
    .accountList:
        [ a list of TccAccount instances representing the qualifying
          accounts from TCC LDAP ]
    .get():
        [ if no entry in self is selected ->
            return None
        else ->
            return a TccAccount instance representing the selected
            entry in self ]
    .lookupUid(uid):
        [ if uid matches the user ID of a TccAccount instance in self ->
            return that TccAccount
        else -> raise KeyError ]
    .clear():
        [ self := self with all widgets cleared ]
```
32.2. uidfinder.py: Imports

```python
# - - - - - I m p o r t s

The test driver uses the Python 3.x-style print() function. Python requires that all imports from the __future__ module precede all other executable statements.

```from __future__ import print_function```

The string module provides various text services.

```python
import string
```

The Tkinter module provides most of the GUI infrastructure. We also use the ttk “themed widget” extensions. The tkMessageBox provides services for popup error messages.

```python
import Tkinter as tk
import ttk
import tkMessageBox
```

For the LDAP interface to Python, see the web site for the LDAP client API for Python.

```python
import ldap
```

The logic that does a name search comes from a separate project, namesearch.py: A name search module for Python.

```python
import namesearch
```

32.3. uidfinder.py: Manifest constants

```python
# - - - - - M a n i f e s t c o n s t a n t s

Names of constants are capitalized with “_” as the word separator.

32.3.1. UF_CLEAR_BUTTON

Label on the Clear button.

```python
UF_CLEAR_BUTTON = "Clear"
```

32.3.2. UF_FIND_UID_BUTTON

Text on the Find user ID button.

---

19 http://www.python-ldap.org/
20 http://www.nmt.edu/tcc/projects/namesearch/
32.3.3. UF_NAME_SEARCH_BUTTON
Text on the Name search button.

```python
UF_NAME_SEARCH_BUTTON = "Name search"
```

32.3.4. BOX_FONT
Font for the lines in the main Listbox. In ttk widgets this is done through modifying the base style, but we still need an explicit font for Listbox because there is no ttk version of that widget.

```python
BOX_FONT = ('DejaVu Sans Mono', 9)
```

32.3.5. BUTTON_FONT

```python
BUTTON_FONT = ('Helvetica', 14, 'bold')
```

32.3.6. ENTRY_FONT

```python
ENTRY_FONT = ('Helvetica', 12)
```

32.3.7. LABEL_FRAME_FONT

```python
LABEL_FRAME_FONT = ('Helvetica', 40)
```

32.3.8. UID_ENTRY_WIDE
Width of the entry field for UID.

```python
UID_ENTRY_WIDE = 9
```

32.3.9. BOX_LINE_HIGH
Number of lines in the main Listbox.

```python
BOX_LINE_HIGH = 10
```

32.3.10. BOX_LINE_WIDE
Length of each line in the Listbox. In a survey of names in April 2013, the longest name was about 35 characters. Add 2 for the type code and 8 for the UID, plus 4 for padding, and 50 should do it.
32.3.11. **LDAP_FILTER_ALL**
For selecting all accounts in an LDAP search.

```
LDAP_FILTER_ALL = '(objectClass=*)'
```

32.3.12. **LDAP_UID**
Name of the user ID attribute in LDAP.

```
LDAP_UID = 'uid'
```

32.3.13. **LDAP_ACCT_TYPE**
Name of the account type attribute in LDAP.

```
LDAP_ACCT_TYPE = 'tccAccountType'
```

32.3.14. **LDAP_CN**
Common name attribute in LDAP.

```
LDAP_CN = 'cn'
```

32.3.15. **LDAP_ATTR_LIST**
List of the attribute names we want from LDAP.

```
LDAP_ATTR_LIST = [LDAP_UID, LDAP_ACCT_TYPE, LDAP_CN]
```

32.3.16. **LDAP_SERVER**
Name of the TCC LDAP server.

```
LDAP_SERVER = 'ldaps://ldap0.nmt.edu:636/
```

32.3.17. **ACCOUNTS_DN**
The Distinguished Name of the node that contains all TCC accounts.

```
ACCOUNTS_DN = 'ou=accounts,dc=tcc,dc=nmt,dc=edu'
```
32.3.18. ACCT_MAP

In this dictionary, the keys are the possible values of the LDAP_ACCT_TYPE. For account types that are eligible for Moodle accounts, the corresponding value is the two-letter account type code to be displayed in the listbox; for ineligible account types, the value is None.

Note

This list must agree with the constant QUALIFYING_ACCT_TYPES in the cmsaccs script documented in cmsimport2: Courseware Banner integration tools. At this writing (April 2013), the definition looks like this:

```python
QUALIFYING_ACCT_TYPES = [ 
    "DistanceEd", "Faculty", "Staff (Payroll)", "Staff (non-payroll)", 
    "Graduate", "MST", "Undergraduate", "Special" ]
```

```python
ACCT_MAP = {
    '?': None,
    'Alumni': None,
    'Alumni Basic': None,
    'Club': None,
    'CommunityCollege': None,
    'Computer Account': None,
    'Department': None,
    'DistanceEd': 'DE',
    'Emeritus': None,
    'Faculty': 'Fa',
    'Graduate': 'Gr',
    'MST': 'MS',
    'Machine Account': None,
    'NRAO': None,
    'Outside': None,
    'Research': None,
    'Software': None,
    'Special': 'Sp',
    'Spouse': None,
    'Staff (Payroll)': 'St',
    'Staff (non-payroll)': 'Sn',
    'TCC Internal': None,
    'Temporary': None,
    'Undergraduate': 'UG',
}
```

32.3.19. SET_OF_ALL_LETTERS

A set containing all the letters in ASCII.

```python
SET_OF_ALL_LETTERS = set(string.letters)
```

---

http://www.nmt.edu/tcc/doc/ibs/cmsimport2/
32.4. uidfinder.py: Specification functions

```python
# - - - - - S p e c i f i c a t i o n f u n c t i o n s

Specification functions are part of the Cleanroom protocol, and serve as notational shorthand. All such names are multiple words separated by hyphens, so they cannot be mistaken for Python names.

The acct-display function shows how the lines in the listbox are formatted.
```

```python
# - - - a c c t - d i s p l a y
#
# acct-display(tccAccount, asName) ==
# (the two-letter account type code ACCT_MAP[tccAccount.acctType]) +
# (tccAccount.uid) + asName
#
```

32.5. class UidFinder: The name lookup widget

```python
# - - - - - c l a s s U i d F i n d e r

class UidFinder(ttk.LabelFrame):
    '''Compound widget for searching for proper names from LDAP.

    State/Invariants:
    .accountMap:
        [ a dictionary whose keys are the UIDs of qualifying
          accounts from TCC LDAP, and each related value is a
          TccAccount instance representing that account ]
    .matchedAccounts:
        [ a list of TccAccount instances corresponding to the
          lines displayed in self._box ]

    Widgets and control variables:
    .uidEntry:
        [ a ttk.Entry for searching by uid ]
    .uidVar:
        [ the tk.StringVar for self._uidEntry ]
    .uidGoButton:
        [ a ttk.Button to initiate UID search ]
    .clearButton:
        [ a ttk.Button that clears self._uidVar and self._searchVar ]
    .searchEntry:
        [ a ttk.Entry for string name search ]
    .searchVar:
        [ the tk.StringVar for self._searchEntry ]
    .searchGoButton:
        [ a ttk.Button to initiate string name search ]
    .boxFrame:
        [ a ttk.Frame to hold self._box and self._boxVScroll ]
```
32.6. UidFinder.__init__()

```python
# - - - U i d F i n d e r . _ _ i n i t _ _

def __init__(self, parent, title, callback=None, accountList=None):
    '''Constructor.''
    
    #-- 1
    # [ self := self as a ttk.LabelFrame with text=title ]
    ttk.LabelFrame.__init__(self, parent, text=title,
                            padding=5)

    #-- 2
    self.title = title
    self.callback = callback
    self.accountList = []
    self.accountMap = {}
    self.matchedAccounts = []

For the widget setup, see Section 32.7, “UidFinder._createWidgets(): Create all widgets and
control variables” (p. 184). The logic that loads up the list of accounts is Section 32.8, “UidFinder._pop-
ulate(): Find all TCC accounts that can have Moodle accounts” (p. 185).
```
#-- 4
# [ if accountList is None ->
#    self.accountList := as invariant from TCC LDAP
#    self._accountMap := as invariant
#  ]
else ->
#    self.accountList := accountList
#    self._accountmap := as invariant ]
self._populate(accountList)

32.7. UidFinder._createWidgets(): Create all widgets and control variables

The handler for the Find user ID button is Section 32.11, “UidFinder._findByUid(): Look up a specific user ID” (p. 188). We also bind the Enter key to the same handler. The button is there just for users who don’t realize they can get the same effect by pressing Enter.

```python
# - - - U i d F i n d e r . _ c r e a t e W i d g e t s

def _createWidgets(self):
    '''Create all control variables and widgets
    '''
    rowx = colx = 0
    self._uidVar = tk.StringVar()
    self._uidEntry = ttk.Entry(self,
                              font=ENTRY_FONT,
                              width=UID_ENTRY_WIDE,
                              textvariable=self._uidVar)
    self._uidEntry.grid(row=rowx, column=colx,
                        padx=5, pady=5, sticky=tk.E)
    self._uidEntry.bind("<KeyPress-Return>", self._findByUid)

    colx += 1
    self._uidGoButton = ttk.Button(self,
                                   text=UF_FIND_UID_BUTTON,
                                   command=self._findByUid)
    self._uidGoButton.grid(row=rowx, column=colx,
                            padx=5, pady=5, sticky=tk.W)

    The handler for the Clear button is Section 32.10, “UidFinder._clearHandler()” (p. 187).
```
```

```python
# - - -
colx += 1
self._clearButton = ttk.Button(self,
                                text=UF_CLEAR_BUTTON,
                                command=self._clearHandler)
self._clearButton.grid(row=rowx, column=colx,
                        padx=5, pady=5)

rowx, colx = rowx+1, 0
self._searchVar = tk.StringVar()
self._searchEntry = ttk.Entry(self, width=30,
                              font=ENTRY_FONT,
                              textvariable=self._searchVar)
```
self._searchEntry.grid(row=rowx, column=colx, padx=5, pady=5, columnspan=2, sticky=tk.E)

See Section 32.14, “UidFinder._nameSearch(): Generic name search” (p. 189).

colx += 2
self._searchButton = ttk.Button(self, text=UF_NAME_SEARCH_BUTTON, command=self._nameSearch)
self._searchButton.grid(row=rowx, column=colx, padx=5, pady=5, sticky=tk.W)
self._searchEntry.bind("<KeyPress-Return>", self._nameSearch)

rowx, colx = rowx+1, 0
self._boxFrame = ttk.Frame(self)
self._boxFrame.grid(row=rowx, column=colx, columnspan=3, padx=5, pady=5)

See Section 32.15, “uidfinder.py: UidFinder._selectHandler(): Handle a click on a line in the listbox” (p. 190), which is called when the user clicks on a line in the listbox.

#-#-# Widgets inside self._boxFrame 
#-#-# rowx, colx = 0, 0
self._box = tk.Listbox(self._boxFrame, selectmode=tk.SINGLE, # Only allow one line to be selected
font=BOX_FONT,
width=BOX_LINE_WIDE, height=BOX_LINE_HIGH)
self._box.grid(row=rowx, column=colx)

colx += 1
self._boxVScroll = ttk.Scrollbar(self._boxFrame, orient=tk.VERTICAL, command=self._box.yview)
self._boxVScroll.grid(row=rowx, column=colx, sticky=tk.N+tk.S)
self._box['yscrollcommand'] = self._boxVScroll.set
self._box.bind("<Button-1>", self._selectHandler)

32.8 UidFinder._populate(): Find all TCC accounts that can have Moodle accounts

#-#-# U i d F i n d e r _ p o p u l a t e
def _populate(self, accountList):
    """Find all the appropriate TCC accounts.

    [ accountList is a list of TccAccount instances or None ->
    if accountList is None ->
        self.accountList := as invariant from TCC LDAP
        self._accountMap := as invariant
If a previous instantiation already build the list of qualified TCC accounts, use that; otherwise see Section 32.9, “UidFinder._ldapQuery()” (p. 186).

```python
#-- 1
# [ if accountList is None ->
#   self.accountList := as invariant from TCC LDAP
# else ->
#   self.accountList := accountList ]
if accountList is None:
    self.accountList = self._ldapQuery()
else:
    self.accountList = accountList
```

Now build the dictionary mapping user IDs to TccAccount instances.

```python
#-- 2
# [ self._accountMap := as invariant from self.accountList ]
for tccAccount in self.accountList:
    self._accountMap[tccAccount.uid] = tccAccount
```

### 32.9. UidFinder._ldapQuery()

```python
#-- - - U i d F i n d e r . _ l d a p Q u e r y

def _ldapQuery(self):
    '''Build and return the set of qualified TccAccounts.

    [ return a list of TccAccount instances representing
    qualifying accounts from TCC LDAP ]
    ...
    '''

The three attributes we need are all available through an anonymous LDAP binding, so we don’t need any special authentication.

```python
#-- 1
# [ binding := an anonymous binding to LDAP_SERVER
# accountList := a new, empty list ]
binding = ldap.initialize(LDAP_SERVER)
binding.simple_bind()
accountList = []
```

The binding.search_s() method returns a list of 2-tuples, the second elements of which are dictionaries mapping attribute names to lists of attribute values.
The constructor of the TccAccount class turns these attribute dictionaries into instances, which we store in two places. The .accountList attribute is a plain list, to be used when we do name searches. The .accountMap dictionary is for lookup by UID.

```python
for attr in attrList:
    # attr is a dictionary whose keys are the elements of LDAP_ATTR_LIST and each related value is a list of attribute values ->
    # self._accountMap +=: an entry whose value is a TccAccount instance made from attr and the corresponding key is its .uid attribute
    # self.accountList +=: that TccAccount instance
    tccAccount = TccAccount(attr)
typeCode = ACCT_MAP[tccAccount.acctType]
    if typeCode is not None:
        accountList.append(tccAccount)

return accountList
```

### 32.10. UidFinder._clearHandler()

Clears the two search fields and the list.

```python
def _clearHandler(self):
    '''Clear the text entries.
    '''
    self.uidVar.set('')
    self.searchVar.set('')
    self._clearList()
```
32.11. `UidFinder._findByUid()`: Look up a specific user ID

```python
# --- U i d F i n d e r . _ f i n d B y U i d

def _findByUid(self, event=None):
    """Look up the uid in self._uidEntry."
    
    #-- 1
    # [ if the value of self._uidVar is a key in
    #   self._accountMap ->
    #   self := self displaying a line representing the
    #   self._accountMap[that value]
    #   self._matchedAccounts := a one-element list
    #   containing self._accountMap[that value]
    #   self._searchVar := ''
    # else ->
    #   self._box := one line displaying an error message
    #   self._matchedAccounts := a new, empty list ]
    uid = self._uidVar.get().strip()
    try:
        tccAccount = self._accountMap[uid]
        self._clearList()
        self._listAdd(tccAccount)
        self._box.selection_set(0)
        self._searchVar.set('')
    except KeyError:
        self._clearList()
        message = ("?? {0:<8s} *** Unknown user ID "
                   "***.format(uid))
        self._box.insert(tk.END, message)
```

32.12. `UidFinder._clearList()`: Clear out the list

```python
# --- U i d F i n d e r . _ c l e a r L i s t

def _clearList(self):
    """Make self._box empty."

    [ self._matchedAccounts := a new, empty list
      self._box := empty ]

    self._matchedAccounts = []
    self._box.delete(0, tk.END)
```

32.13. `UidFinder._listAdd()`: Add one entry to the list

Because we expect the list of entries to have been sorted into display order, all we need to populate the
listbox is this method, which adds a new last line.
def _listAdd(self, tccAccount, altName=None):
    '''Add a new account line at the end of the listbox.

    [ (tccAccount is a TccAccount instance) and
      (altName is a full-name or None) ->
        self._box := self._box with a text display of tccAccount
    added as the new last line, using altName instead of
      tccAccount.fullName if it is provided ]
    ...

    displayName = (tccAccount.fullName
      if altName is None
      else altName)
    typeCode = ACCT_MAP[tccAccount.acctType]
    if typeCode is None:
        typeCode = '-'
    text = ("{0:<2s} {1:<9s} {2}".format(
        typeCode, tccAccount.uid, displayName))
    self._box.insert(tk.END, text)
    self._matchedAccounts.append(tccAccount)

 32.14. UidFinder._nameSearch(): Generic name search

First we retrieve the search term from self._searchEntry and remove leading and trailing whitespace.
If the term doesn’t contain at least one letter, flag it as an error. If it looks okay, create a NameSearch
instance that will apply the rules for name matching, clear the listbox, and also clear out
self._uidEntry so that it will be clear to the user that the listbox’s content came from the Name search
button and not the Find user id button.

#-- 1
# [ target := contents of self._searchEntry without
#     leading and trailing whitespace ]
target = self._searchVar.get().strip()

#-- 2
# [ if target contains no letters ->
#    (desktop) := an error popup
#    return
#  else ->
#    searcher := a namesearch.NameSearch instance that
#    matches target
#    self._box := empty
#    self._uidVar := ''
We feed the NameSearch instance each name from the complete list using its .search() generator method. This method generates an instance of class namesearch.NameMatch for each location in the source line where there is a match.

The NameMatch.__str__() method renders it as displayed, possibly rotated so that the match is at the start of the line. So we can apply the str() function to the instance to get the name part of the line.

The NameMatch.__cmp__() methods orders instances in case-insensitive alphabetical order as displayed. However, we must be able to trace each displayed line in the listbox back to the corresponding user ID as a TccAccount instance.

Thus, the next step is to build a list of tuples \((m, a)\) where \(m\) is a NameMatch instance and \(a\) is the corresponding TccAccount. Then, after sorting this list, we rebuild the listbox from the sorted list, storing the TccAccount instances in self._matchedAccounts.

Later, when the user clicks on some line at position \(k\) in the listbox, we know that the corresponding TccAccounts instance is self._matchedAccounts[k].

32.15. uidfinder.py: UidFinder._selectHandler(): Handle a click on a line in the listbox

This handler is called when the user clicks inside the listbox. If the constructor was passed a callback, that function is to be called with the TccAccount instance corresponding to the click location.
The `tk.Listbox.nearest()` method will tell us the line closest to the click position, but in a partially filled listbox, any click below the last line, the `.nearest()` method points at the last line. Here, we discard any click that was not in the actual bounding box.

```python
# - - - U i d F i n d e r . _ s e l e c t H a n d l e r
def _selectHandler(self, event):
    '''On Button-1 inside the listbox, call the callback.
    
    [ if (self.callback is None) or
      (event.y is not inside the line indicated by
      self._box.nearest(event.y) ->
    I
    else ->
    call self.callback(t) where t is the TccAccount
    instance corresponding to the line indicated by
    self._box.nearest(event.y) ]
    ...
    #-- 1
    if self.callback is None:
        return

    #-- 2
    # [ nearx := the index of the visible line closest to
    #   event.y ]
    nearx = self._box.nearest(event.y)

    #-- 2
    # [ if event.y is not inside the bounding box of line
    #   (nearx) ->
    #   return
    #   else ->
    #   call self.callback(self._matchedAccounts[nearx]) ]
    xmin, ymin, wide, high = self._box.bbox(nearx)
    xmax = xmin + wide - 1
    ymax = ymin + high - 1
    if ((xmin <= event.x <= xmax) and
        (ymin <= event.y <= ymax)):
        self.callback(self._matchedAccounts[nearx])
```

### 32.16. uidfinder.py: UidFinder.get()

```python
# - - - U i d F i n d e r . g e t
def get(self):
    '''Return the selected line as a TccAccount, or None.
    ...'''
```

The `Listbox.curselection()` method returns a tuple containing the positions of selected lines, counting from 0. Assuming the widget was created with `selectmode=tk.SINGLE`, this tuple will contain the index of the selected line; if no line is selected, the tuple will be empty.
uidfinder.py

```python
#-- 1
# [ if a line in self._box is currently selected ->
#   linex := position of the first selected line
# else -> return None ]
selectTuple = self._box.curselection()
if len(selectTuple) == 0:
    return None
else:
    linex = int(selectTuple[0])
    return self._matchedAccounts[linex]
```

32.17. uidfinder.py: UidFinder.lookupUid()

```python
#-- - - U i d F i n d e r . l o o k u p U i d

def lookupUid(self, uid):
    '''Translate a user ID to a TccAccount.'''
    return self._accountMap[uid]
```

32.18. uidfinder.py: UidFinder.clear()

```python
#-- - - U i d F i n d e r . c l e a r
def clear(self):
    '''Clear all self's widgets.'''
    #-- 1
    # [ self._uidVar := empty
    # self._searchVar := empty
    # self._box := empty ]
    self._uidVar.set('')
    self._searchVar.set('')
    self._box.delete(0, tk.END)
```

32.19. uidfinder.py: class TccAccount

An instance of this class holds information about one user ID from the TCC LDAP.

```python
#-- -- - - c l a s s   T c c A c c o u n t

class TccAccount(object):
    '''Represents one TCC account.'''
    
    Exports:
    TccAccount(attrSet):
        [ attrSet is an element of the list returned by search_s ->
```
return a TccAccount instance representing attrSet

.. uid: [uid as a str]
.. acctType: [tccAcctType as a str]
.. fullName: [accountholder's name as a str]

.. _cmp_(self, other):
   [primary key .fullName, secondary key .uid, ascending order]

.. _str_(self): [return a string rendering of self]

32.20. uidfinder.py: TccAccount.__init__()

--- TccAccount.__init__

```python
def __init__(self, attrSet):
    '''Constructor.''
    #-- 1
    # [ self.uid := first element of attrSet[LDAP_UID]
    #   self.acctType := first element of attrSet[LDAP_ACCT_TYPE],
    #   defaulting to '?' ]
    self.uid, = attrSet[LDAP_UID]
    self.acctType, = attrSet.get(LDAP_ACCT_TYPE, ['?'])
```

In most LDAP entries, the `cn` is a string of the form "uid full-name", where full-name has the form "given [middle] family". However, some older and internal entries don't follow this convention, but that's okay because none of them should be Moodle-qualified accounts.

```python
#-- 2
# [ if attrSet[LDAP_CN] is a single word ->
#   self.fullName := that word
# else ->
#   self.fullName := all words after the first ]
rawName, = attrSet[LDAP_CN]
wordList = rawName.split(None, 1)

if len(wordList) < 2:
    self.fullName = wordList[0]
else:
    self.fullName = wordList[1]
```

32.21. uidfinder.py: TccAccount.__cmp__()

This method defines the ordering relation when sequences of TccAccount instances are sorted. The primary key is the full name, with the UID as a tiebreaker.

```python
--- TccAccount.__cmp__

def __cmp__(self, other):
    '''Order by (name, uid)'''
```
return cmp((self.fullName, self.uid),
            (other.fullName, other.uid))

32.22. uidfinder.py: TccAccount.__str__()

```python
# - - - T c c A c c o u n t . _ _ s t r _ _

def __str__(self):
    """Render self as a string."
    return (<TccAccount({t.uid!r}, {t.acctType!r}, 
                        "{t.fullName!r}">
               .format(t=self))
```

32.23. uidfinder.py: Test driver

```python
# - - - - - t e s t m a i n

def main():
    """Test driver."
    app = TestApp()
    app.master.title("uidfinder test main")
    app.mainloop()

# - - - - - T e s t A p p

class TestApp(ttk.Frame):
    """Test driver for uidfinder.

    Widgets:
    0 1 2
    +-------------+--------------+--------------+
    0 | ._uidFinder | ._readButton | ._quitButton |
    +-------------+--------------+--------------+
    ""
    def __init__(self):
        """Constructor."
        ttk.Frame.__init__(self)
        self.grid()
        self._restyle()
        self._createWidgets()

    def _restyle(self):
        """Modify the base widget classes."
        sB = ttk.Style()
```
sB.configure('TButton', font=BUTTON_FONT)

sE = ttk.Style()
sE.configure('TEntry', font=ENTRY_FONT)

sLF = ttk.Style()
sLF.configure('TLabelFrame',
    padding='10',
    font=LABEL_FRAME_FONT)

def _createWidgets(self):
    '''Create and grid all widgets.'''
    rowx = colx = 0
    self._uidFinder = UidFinder(self, "Find a name",
        callback=self._clickHandler)
    self._uidFinder.grid(row=rowx, column=colx,
                        padx=5, pady=5)

    colx += 1
    self._readButton = ttk.Button(self,
        command=self._readHandler,
        text="Read")
    self._readButton.grid(row=rowx, column=colx, sticky=tk.N)

    colx += 1
    self._quitButton = ttk.Button(self,
        command=self.quit,
        text="Quit")
    self._quitButton.grid(row=rowx, column=colx, sticky=tk.N)

def _clickHandler(self, tccAccount):
    '''The user clicked on a line of the listbox.'''
    print("Click:", tccAccount)

def _readHandler(self):
    '''The user clicked the Read button.'''
    whichAccount = self._uidFinder.get()
    if whichAccount is None:
        print("No name is currently selected.")
    else:
        print("Current selection is", whichAccount)

32.24. uidfinder.py: Epilogue
This code runs the test main if the file is executed (as opposed to being imported).

# - - - - - Epilogue
33. Testing

UI testing is necessarily labor-intensive. This section describes the author’s own set of tests. Naturally an author will have the same blind spots while testing that they had in design, so these tests do not claim full coverage. The initial subsections refer to Section 3.2, “User stories” (p. 10); later subsections address checking for anticipated error conditions. (Testing for unanticipated error conditions is a good trick.)

The tester will want to run these tests in a directory separate from a production directory, since the cmsadds application uses a number of specific named files.

A number of steps request that you examine the adds-season-yyyy.xml file. As this is raw XML with no particular line structure, you may find xxe useful; see Building XML documents with XXE22.

33.1. Tests for user story A

See Section 3.2.1, “Story A: Single section, single instructor” (p. 10).

1. In an empty directory, bring up cmsadds with no command line arguments and check that it picks the correct default semester. Note the set of semesters available, and bring it up again for each of these semesters, using command line arguments.

2. Technically, in story A you know the TCC UID of the primary instructor, but if you can’t think of a UID with an account type that gets a Moodle account, use the Name search field to find one.

   Once you know the UID of a Moodle-qualified account, enter it in the Find user ID field. To execute a search, you should be able to use either Enter in that field, or click on Name search. This will set the name list to a single line showing the account type code, UID, and full name of that person, and that line will be highlighted.

3. Click on that highlighted line. This will bring up the primary instructor panel in the southwest corner of the application. Make sure that the listboxes for sections and metacourses are clear, and the instructor’s name is displayed in the three fields.

4. Change all three of the name fields (given, middle, and family). Click the Save adds-season-yyyy.xml button, and examine the adds-season-yyyy.xml file to see if your changes were saved.

5. Click Close and insure that the instructor panel is cleared. Then click the instructor’s name again and insure that the name is as you edited it.

6. Now we’ll exercise the department selector. Hold down mouse button 1 and insure that the correct department codes and names appear in the drop-down menu.

7. Find a department that has courses in the current semester, select it, and insure that it displays all the sections correctly. Verify the section list against the official Banner schedule page.

8. Find a department that has no courses in the current semester, select it, and insure that the error popup is apt.

22 http://www.nmt.edu/tcc/help/xxe/
9. Test that you can enter a department code in the Find department field, and bring up its schedule (or the error popup if it has no sections) using either by pressing Enter in the field or by clicking Find department.

10. Enter a nonsense department code and insure that you get an apt popup.

11. Next we'll add a section to the instructor. Select a department that has sections in the current semester, highlight a section, and click Add selected section to this instructor. Insure that the section appears in the listbox under Sections: right-click to edit.

Click Quit. You should get a popup warning you that the changes have not been saved, and asking if you want to discard your changes. Before clicking either choice, examine the adds-season-yyyy.xml file; then click No and check the file to be sure that the section you just added is not there.

12. Restart cmsadds and go through the above steps to add a different section. Click Quit and this time answer Yes. Check the adds-season-yyyy.xml file to insure that the instructor and section are there.

33.2. Tests for user story B

In this story, you will create a metacourse.

Procedure 1. Metacourse creation

1. Bring up cmsadds and select an instructor.

2. Bring up a department schedule for some department like MATH that has lots of sections every semester.

3. Add two or more sections from that department to the instructor.

4. In the Metacourses: right-click to edit area, click Add metacourse. A line containing “?” will appear in the metacourse list in the instructor panel. The schedule panel will disappear and be replaced by a metacourse panel, also showing “?” as the course title.

5. In the Metacourses panel, edit the Title field so that it is empty. Then click anywhere else to move focus away from that field, and insure that a relevant popup appears. Dismiss the popup.

6. In the Title field, enter a metacourse title. Click anywhere else to cause to move focus away from that field, and insure that the new title appears in the metacourse list of the instructor panel.

7. Before adding any child sections, click the Close button on the Metacourses panel. You will get a popup with two choices: Discard this metacourse and Return to editing this metacourse. First, click Return to editing this metacourse and insure that leaves you editing the metacourse panel.

Then again click Close and this time choose Discard this metacourse. The transient panel should blank and the metacourse title should disappear from the metacourse list in the instructor panel.

At this point Save adds-season-yyyy.xml and inspect the adds-season-yyyy.xml file to be sure the metacourse you deleted is not in that file.

8. Recreate the metacourse and give it a title. Then select and add exactly one section from the instructor's section list and use the Add selected section button to add them to the section list in the metacourse panel.

9. Click the metacourse panel’s Close button and insure that a popup appears complaining that the metacourse doesn’t have enough sections. Click Return to editing this metacourse to return to the metacourse panel.
10. Add another section or two to the metacourse and click the \textit{Save adds-season-yyyy.xml} button. Examine the \textit{adds-season-yyyy.xml} file to be sure the metacourse appears there as it does in the GUI.

\section*{33.3. Tests for user story C}

The new element of this story is that the instructor provides a name without a TCC UID. The \textit{Name search} field is intended to address the need to look up a UID knowing only the person's name or a distinctive part of it.

The test here is to think of someone you know who has a Moodle account and see if you can find their TCC UID using the \textit{Name search} field.

\section*{33.4. Tests for user story D}

New elements here: additional role assignments, and the request to reload the new section from the old one.

\textbf{Procedure 2. Preserving a reload request}

1. Pick an \textit{adds-season-yyyy.xml} file with an instructor that has at least one section, or create such an instructor.

2. Right-click on a section in the \textit{Sections: right-click to edit} list box. This will bring up a section panel on the right.

3. Enter an old XCrn in the \textit{Reload from} field, e.g., “54321.2011”.

4. Click the \textit{Advising Center access} checkbox.

5. In the \textit{Select roleholder} area, pick someone other than the primary instructor, select that person's line by left-clicking on it, and then in the \textit{Add roleholder} area, click the \textit{Instructor} button. A line for the roleholder should appear in the \textit{Roles} list box, showing the code “i” followed by the roleholder’s TCC UID and then their TCC full username.

6. Repeat the previous step to add two more different people in the \textit{Non-editing teacher} and \textit{Grader} roles, and verify that their account type codes (“n” and “g” respectively), account UIDs, and names appear in the \textit{Roles} list box.

7. \textit{Save} and inspect the \textit{adds-season-yyyy.xml} file to insure that the reload request and all four of the roles (including the Advising Center role) appear in the file.

8. Add yet another different person in the \textit{Grader} role. Then select one of the roleholders by selecting on their name inside the \textit{Roles} list box, and click \textit{Delete selected role}. Insure that the selected line disappears from the list box.

9. \textit{Save} and examine the \textit{adds-season-yyyy.xml} file to insure that the set of roleholders there matches the set currently displayed in the application.

\section*{33.5. Tests for user story E}

This user story doesn't really involve any new features of the application; it deals with instructor communication issues.
33.6. Error case testing

These tests are in whatever order the author thought of them.

33.6.1. Deleting an instructor that still has sections

1. Make an instructor with one section. Click on Remove this instructor from the CMS and check that the error message appears and is apt. Cancel out, save, and inspect the adds-season-yyyy.xml file to be sure the instructor’s section is still there.

2. Then click Remove this instructor from the CMS, confirm that you want to discard the data, save, and make sure it’s gone from the file.

33.6.2. What if we can’t save?

Procedure 3. Detection of unwriteable save file

• Create an adds-season-yyyy.xml file and remove its write permission. Then click Save adds-season-yyyy.xml and verify that you get a useful popup.

33.6.3. Archive subdirectory creation

This tests for a defect that was discovered in version 1.0.1. The archive subdirectory is used both for copies of adds-season-yyyy.xml files made after Save adds-season-yyyy.xml and as a place for sects_.xml files made during upload.

As originally written, the same logic was used to save for the Save adds-season-yyyy.xml and for the Upload cms courses button, and this logic insured that the subdirectory exists before saving. However, after some refactoring, if the subdirectory didn’t exist and the user clicked Upload cms courses before Save adds-season-yyyy.xml, the code didn’t insure that the subdirectory existed.

Procedure 4. Archive directory creation during upload

1. This one requires live uploading, so open cmsadds with a file containing valid course requests for a semester in the past, and delete the created courses once the test has passed.

2. Remove the archive subdirectory for the file’s semester.

3. Without saving, click Upload cms courses. Version 1.0.1 failed with a popup complaining that the sects_.xml file couldn’t be moved into the archive subdirectory.

4. After a successful upload, be sure to remove the newly created courses from the Moodle server.