Address Book Example

http://www.cs.gordon.edu/courses/cs211/AddressBookExample/

Introduction

What you will Find Here

This page is the starting point into a series of pages that attempt to give a complete example of object-oriented analysis, design, and programming applied to a small size problem: a simple address book. These pages are similar in style to another, more complicated set of pages I developed earlier: A Simulation of an Automated Teller Machine (ATM). I developed both that set of pages and this in the belief that students would benefit from seeing a *complete* example of OO methodology applied to a single problem. The problem documented by this set of pages is *much* simpler than the ATM Example, and so is more accessible to undergraduate students seeing OO methodology and UML for the first time.

Beginning with a statement of requirements, the process proceeds through analysis, overall design, and detailed design. For this problem, I stopped the process just short of coding, for two reasons:

1. Students are prone to jump right into the coding phase, without spending adequate time on design. This example is meant to focus attention on the all-important design portion of the process.
2. The actual writing of the code is an assignment in two closed labs and an open lab programming project in one of my courses.

(Note: the ATM Example referred to above *does* include complete code.)

Background

The original idea for this example came from a textbook example in the book I used for my Introduction to Programming course - *An Introduction to Object-Oriented Programming with Java* by C. Thomas Wu. The first time I taught the course (in the spring of 2000), I turned this example into a set of labs and a project which culminated in having students develop a fairly complete GUI for the address book (which went far beyond the example in Wu's book). In subsequent years, I have improved the labs and the project, giving my students some design information in the form of various UML diagrams. More recently, I decided that working this up into a complete example of the design process, using UML, could be a valuable teaching tool in my Object-Oriented Software Development course, which many of the students who take the introduction to programming course take the following semester. This example also illustrates the Model-View-Controller and Observer design patterns.

I currently use a vastly-simplified version of these pages for a project in the introduction to programming course.

It may be argued that more diagrams have been used than are really necessary for a project of this size. I wanted to give an example of using a variety of different UML diagrams. Even so, this example does not include any statechart, collaboration, package, activity, component, or deployment diagrams.

Note: Some of the diagrams have been deliberately omitted from the various pages. Students taking my course will no doubt come to understand the phrase "... has been left as an exercise for the reader"!

Using these Pages

Probably the best way to start using these pages is to begin with the requirements and work through the entire analysis, high-level design, and detailed design process.

1. Begin with the Requirements and User Interface document.

   The first task that must be performed in any project is clearly understanding the requirements. This series of pages starts with a statement of the overall requirements for the software, without attempting to discuss the process of actually arriving at them. For a problem of this size, identifying the requirements is fairly straightforward. In a real system, however, identifying the requirements will generally be a non-trivial task. That, however, is not the focus of this set of pages.

2. Then view the Use Cases and Further Analysis.

   Analysis is begun by identifying the use cases that follow from the requirements, and detailing a flow of events for each. Further analysis identifies the key classes that are suggested by the use cases, and considers how each use case can be carried out by an interaction between objects belonging to these classes.

   - The Use Case document has a Use Case Diagram and a series of flows of events, one for each use case. Each use case also has a link to a Sequence Diagram (part of the Design phase) which shows how it is realized; these links can be followed while studying the design phase to see how the analysis phase flows into the design phase.
   - The Further Analysis document deals with both the "big picture" and the details of the various use cases. The former is provided by an Analysis Class Diagram, with each class having a link to its CRC card; the latter by a discussion of how the key objects objects would need to interact in order to implement the use case.

These two documents represent two different ways of viewing the overall system, which continue into the next phase. The Use Case document presents a use-case centric view of the system, focussing on the specific functions it provides. The Further Analysis document presents a class centric view of the system, focussing on how will be built.

3. This example uses CRC Cards and Sequence Diagrams for high level Design. There are certainly other tools that might be used - e.g. the ATM Example referred to above makes use of Collaboration Diagrams and State Charts as well.

    - The responsibilities of each class that arise from the use cases are recorded on a CRC card for each class. The CRC cards could be created by "walking through" each use case, assigning the responsibility for each task to some class.
    - There is a Sequence Diagram for each use case, showing how the use case is realized by interaction of the major objects.

- A Class Diagram shows how the various classes are related to one another. It also shows several additional classes that were "discovered" during the process of creating the Sequence Diagrams - i.e. classes needed to actually build the system, though not evident in the original analysis. On this diagram, the class icon is linked to a detailed design for that class.

4. Detailed design is done by using a detailed UML diagram for each class, showing its attributes and operations. As noted above, although I have not included the actual implementation in Java code, I have included the Javadoc documentation for the classes to "flesh out" the information in the UML diagrams.

5. A demonstration executable version of the the system, in the form of a Java Applet. This version is limited, because the Java mechanisms to protect against malicious code limit access to the file system on the host computer. For this reason, it is not possible to use the "Open", "Save", or "Save As" features of the demonstration - attempting to do so will result in an error dialog.

6. No actual code or complete executable version of this system is provided. The ATM Example System does provide both of these; but since I use the implementation of this system as a programming project, putting the code online would make the project just a bit too simple! :-)

7. A page of maintenance ideas suggests changes that might be made to improve the system. These changes would necessarily involve modifying many of the sample documents, not just modifying code.

Author and Copyright Information

Though the pages are copyrighted, I hereby freely give permission for their reproduction for non commercial educational purposes. I hope they will prove useful to other faculty who are teaching OO methods. I would also really welcome suggestions and feedback - either about the design itself or the way it is presented. Input from UML guru's would especially be appreciated, as I am revising these pages in part as a way to learn UML myself!

Russell C. Bjork
Professor of Computer Science
Gordon College
255 Grapevine Road
Wenham, MA 01984
(978) 927-2300 x 4377
bjork@gordon.edu

Requirements and User Interface for a Simple Address Book

Requirements Statement

The software to be designed is a program that can be used to maintain an address book. An address book holds a collection of entries, each recording a person's first and last names, address, city, state, zip, and phone number.

It must be possible to add a new person to an address book, to edit existing information about a person (except the person's name), and to delete a person. It must be possible to sort the entries in the address book alphabetically by last name (with ties broken by first name if necessary), or by ZIP code (with ties broken by name if necessary). It must be possible to print out all the entries in the address book in "mailing label" format.

It must be possible to create a new address book, to open a disk file containing an existing address book to close an address book, and to save an address book to a disk file, using standard New, Open, Close, Save and Save As ... File menu options. The program's File menu will also have a Quit option to allow closing all open address books and terminating the program.

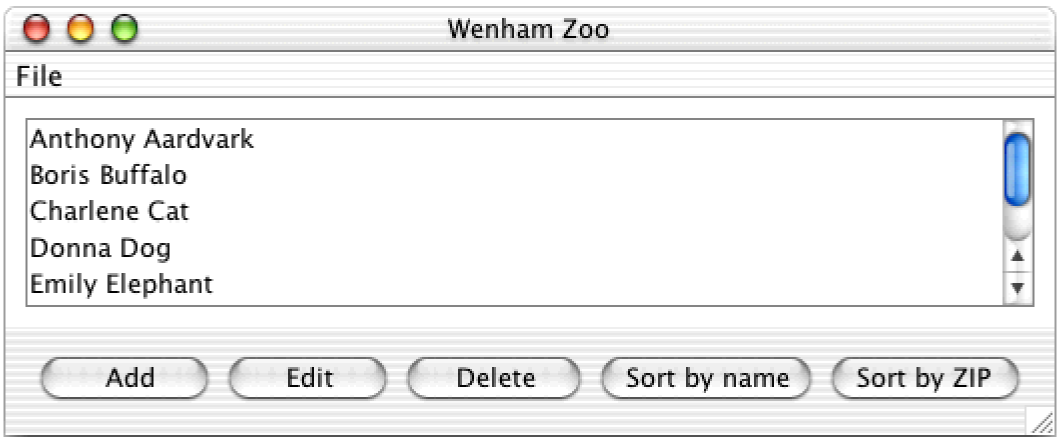
The initial requirements call for the program to only be able to work with a single address book at a time; therefore, if the user chooses the New or Open menu option, any current address book will be closed before creating/opening a new one. A later extension might allow for multiple address books to be open, each with its own window which can be closed separately, with closing the last open window resulting in terminating the program. In this case, New and Open will result in creating a new window, without affecting the current window.

The program will keep track of whether any changes have been made to an address book since it was last saved, and will offer the user the opportunity to save changes when an address book is closed either explicitly or as a result of choosing to create/open another or to quit the program.

The program will keep track of the file that the current address book was read from or most recently saved to, will display the file's name as the title of the main window, and will use that file when executing the Save option. When a New address book is initially created, its window will be titled "Untitled", and a Save operation will be converted to Save As ... - i.e. the user will be required to specify a file.

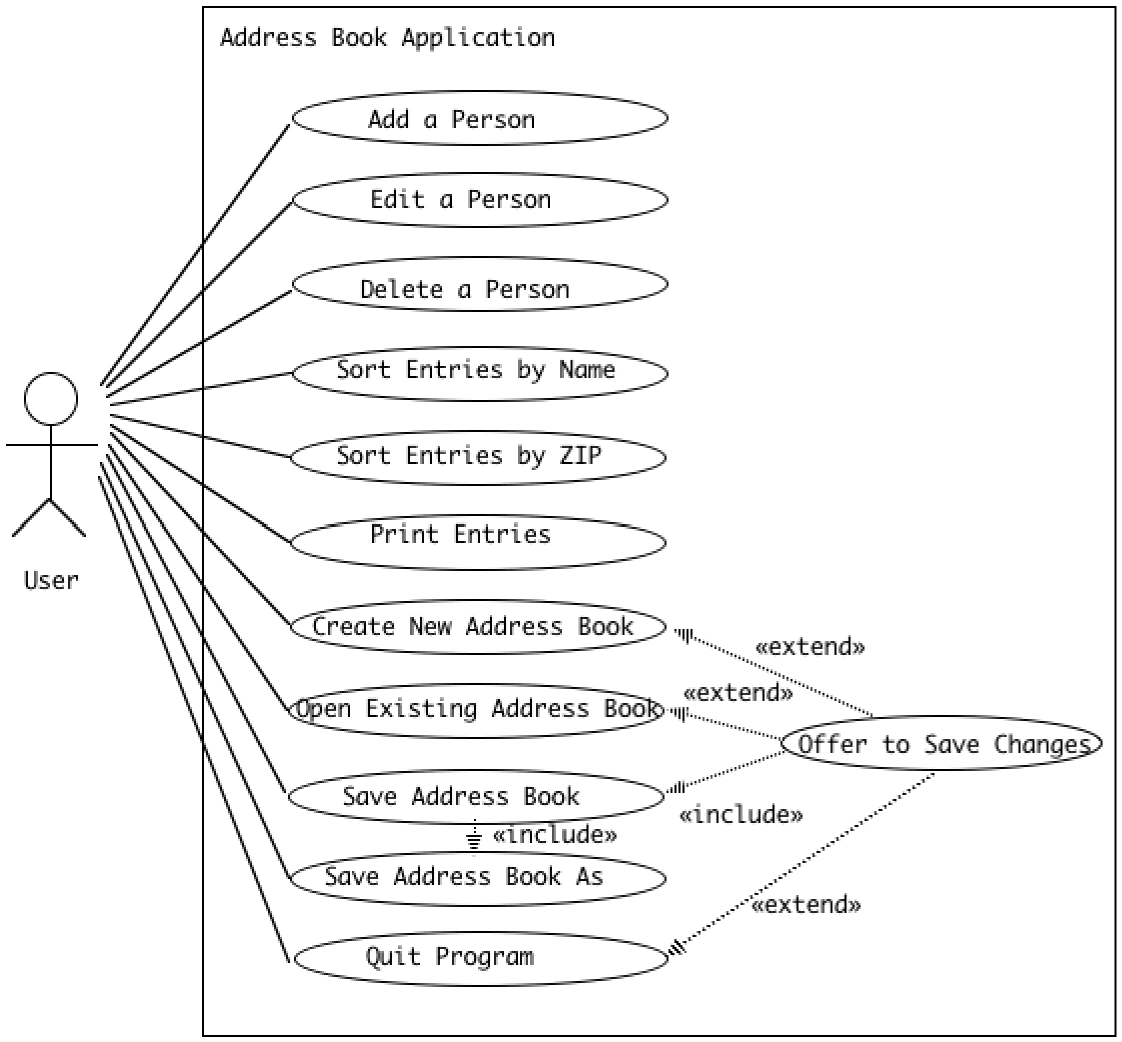
User Interface

Because this is to be a "standard GUI" style application, some attention needs to be given to the user interface at this point. A user interface like the following might be adopted. Not shown in the screen shot is a File menu with New, Open, Close, Save, Save As ..., Print, and Quit options. For the "Edit" and "Delete" buttons, the user must first select a person in the scrolling list of names, and then can click the appropriate button to edit/delete that person.



Use Cases for a Simple Address Book

In the following, use cases are listed in the natural order that a user would think of them. In the actual File menu, items that correspond to the various use cases will be listed in the traditional order, which is slightly different.



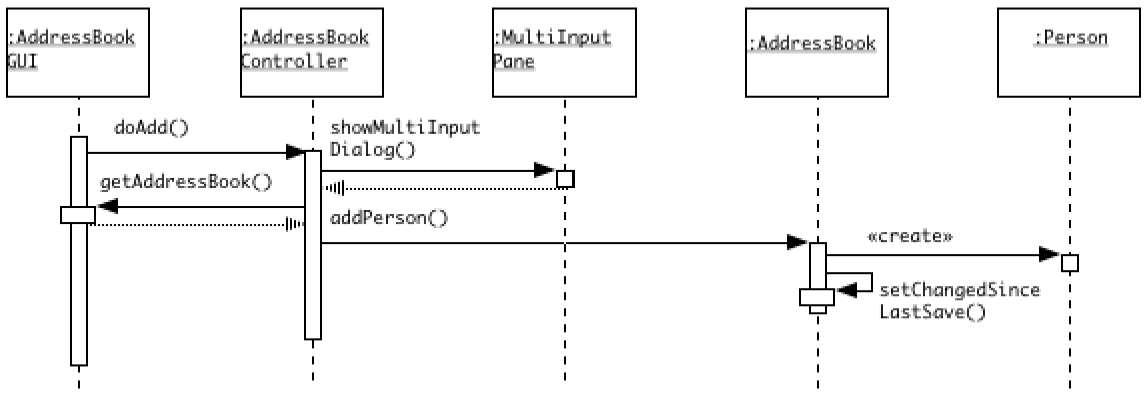
(Click on a use case above to go to the flow of events for that use case)

Flows of Events for Individual Use Cases

Add a Person Use Case

The Add a Person use case is initiated when the user clicks the "Add" button in the main window. A dialog box appears, with title "New Person", containing fields for the user to fill in the new person's first and last names and other information. The box can be dismissed by clicking either "OK" or "Cancel". If the "OK" button is clicked, a new person is added to the end of the address book, and the person's name is added to the end of the list of names in the main window. If the "Cancel" button is clicked, no changes are made either to the address book or to the main window.

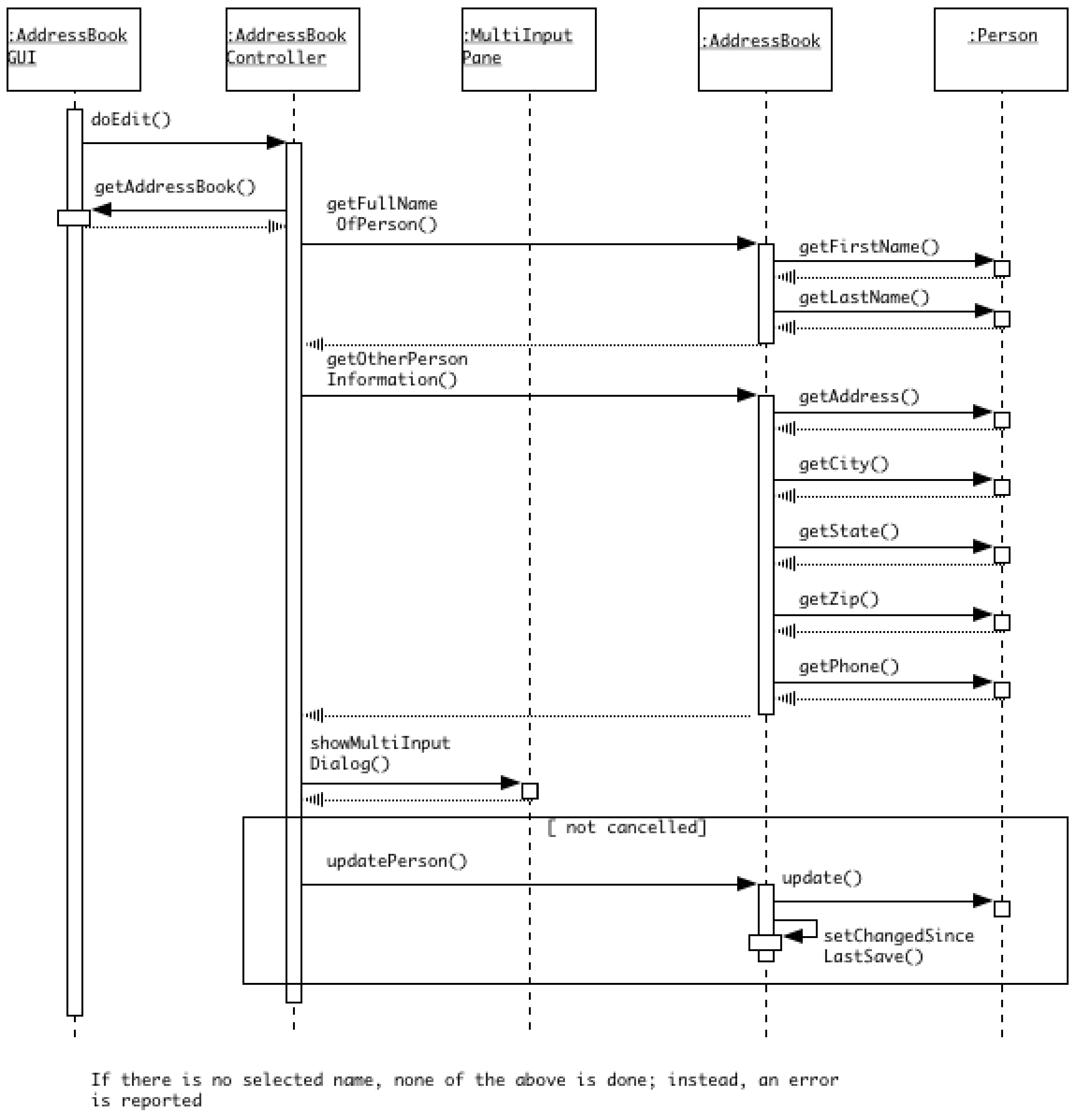
[Sequence Diagram]



Edit a Person Use Case

The Edit a Person use case is initiated when the user either highlights a name in the list of names in the main window and then clicks the "Edit" button, or the user double-clicks a name. In either case, a dialog box, with title "Edit person's name", appears containing current information about the person selected, (except the person's name, which appears only in the title). The user can then edit the individual fields. The box can be dismissed by clicking either "OK" or "Cancel". If the "OK" button is clicked, the entry in the address book for the selected person is updated to reflect any changes made by the user. If the "Cancel" button is clicked, no changes are made to the address book.

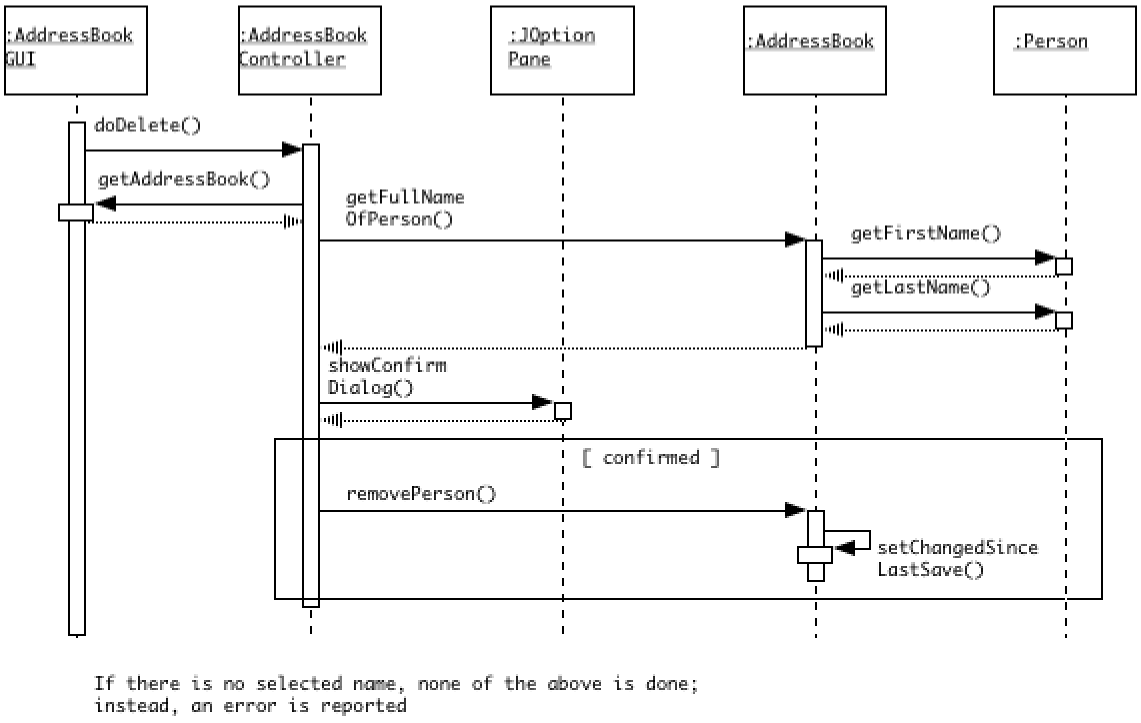
[Sequence Diagram]



Delete a Person Use Case

The Delete a Person use case is initiated when the user highlights a name in the list of names in the main window and then clicks the "Delete" button. A dialog box appears, asking the user to confirm deleting this particular individual. The box can be dismissed by clicking either "OK" or "Cancel". If the "OK" button is clicked, the entry in the address book for the selected person is deleted, and the person's name is deleted from the list of names in the main window. If the "Cancel" button is clicked, no changes are made either to the address book or to the main window.

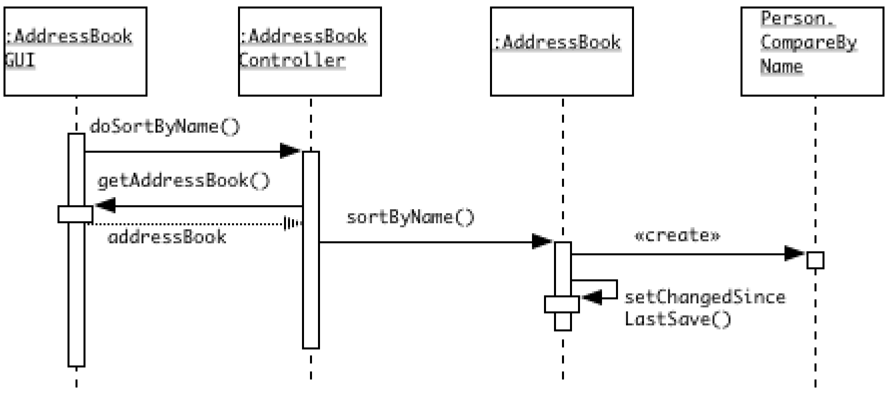
[Sequence Diagram]



Sort Entries by Name Use Case

The Sort Entries by Name use case is initiated when the user clicks the Sort by Name button in the main window. The entries in the address book are sorted alphabetically by name, and the list in the main window is updated to reflect this order as well.

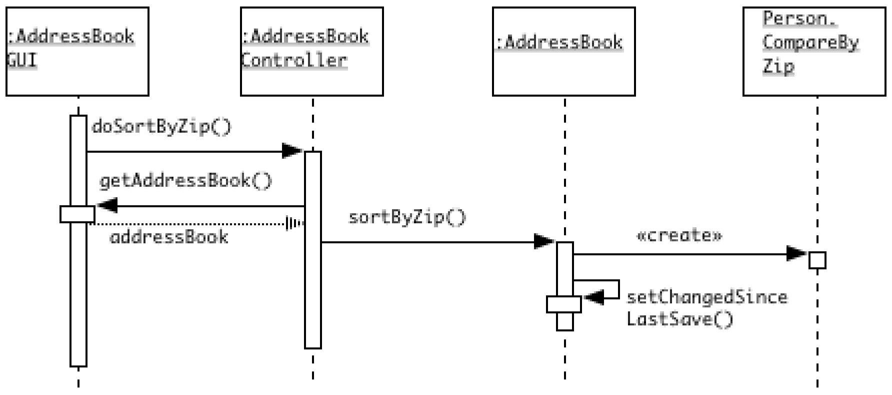
[Sequence Diagram]



Sort Entries by ZIP Use Case

The Sort Entries by ZIP use case is initiated when the user clicks the Sort by ZIP button in the main window. The entries in the address book are sorted by zip code, and the list in the main window is updated to reflect this order as well.

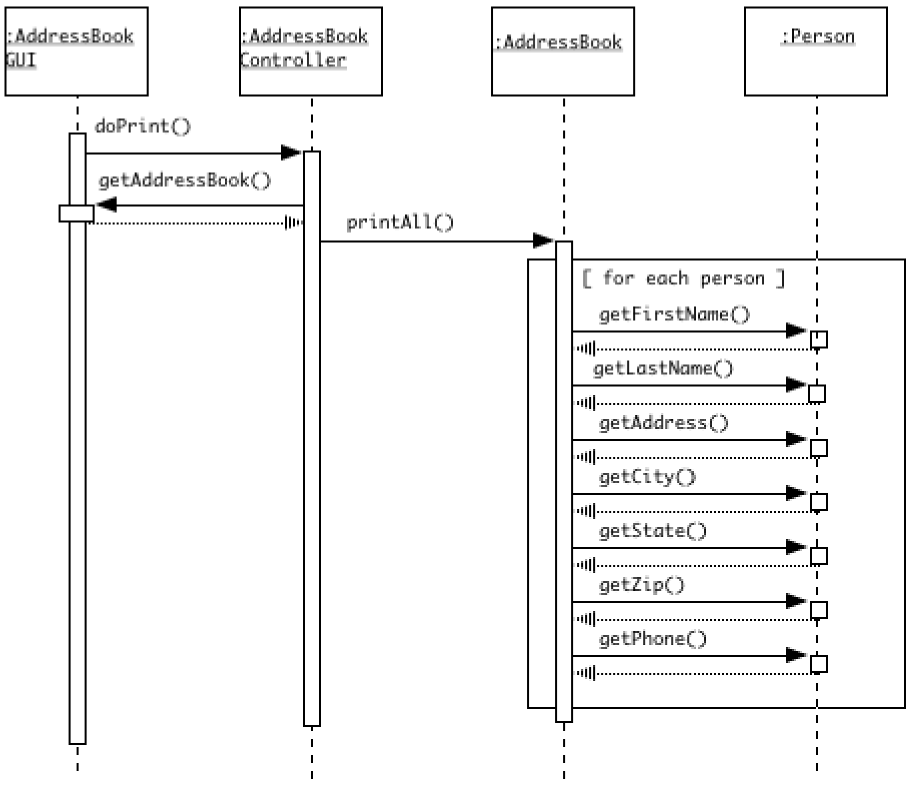
[Sequence Diagram]



Print Entries Use Case

The Print Entries use case is initiated when the user chooses "Print" from the File menu. A save file dialog is displayed, and the user is allowed to choose a file to print the labels to. (If the user cancels the file dialog, the Print operation is canceled.) The current contents of the address book are written out to the specified file (in their current order) in "mailing label" format. No information maintained by the program is changed.

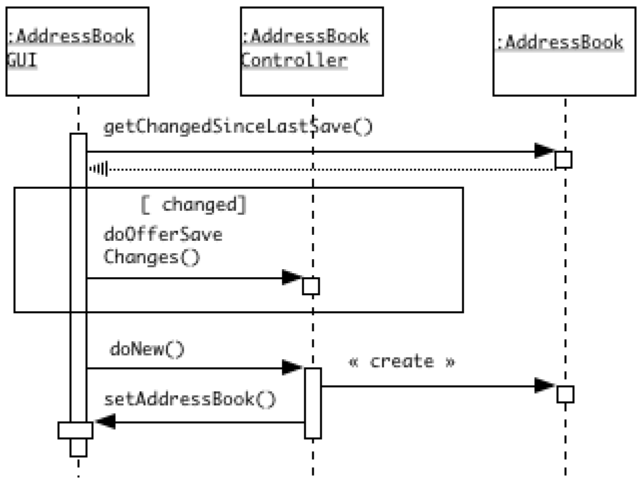
[Sequence Diagram]



Create New Address Book Use Case

The Create a New Address Book use case is initiated when the user chooses "New" from the File menu. If the current address book contents have been changed since the last successful New, Open, Save, or Save As ... operation was done, the Offer to Save Changes extension is executed. Unless the user cancels the operation, a new empty address book is then created and replaces the current address book. This results in the list of names in the main window being cleared, the current file becoming undefined, and the title of the main window becomes "Untitled". (NOTE: These conditions will also be in effect when the program initially starts up.)

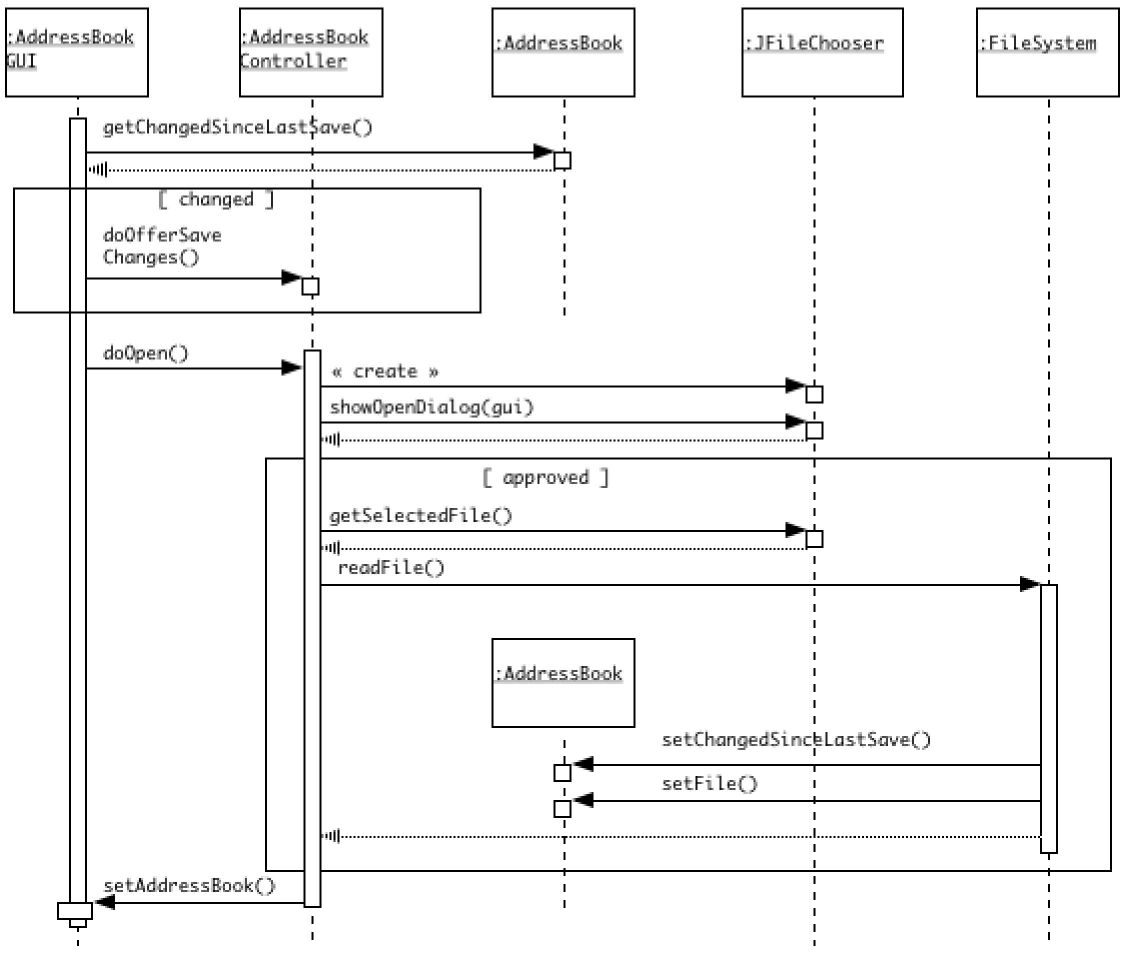
[Sequence Diagram]



Open Existing Address Book Use Case

The Open Existing Address Book use case is initiated when the user chooses "Open" from the File menu. If the current address book contents have been changed since the last successful New, Open, Save, or Save As ... operation was done, the Offer to Save Changes extension is executed. Unless the user cancels the operation, a load file dialog is displayed and the user is allowed to choose a file to open. Once the user chooses a file, the current address book is replaced by the result of reading in the specified address book. This results in the list of names in the main window being replaced by the names in the address book that was read, the file that was opened becoming the current file, and its name being displayed as the title of the main window. (If the user cancels the file dialog, or attempting to read the file results in an error, the current address book is left unchanged. If the cancellation results from an error reading the file, a dialog box is displayed warning the user of the error.)

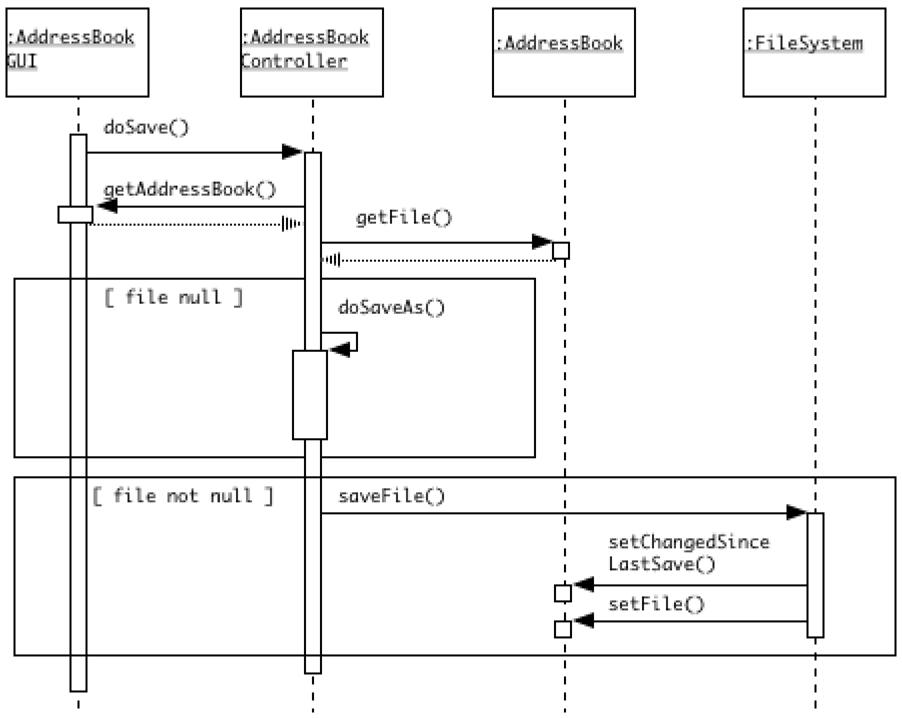
[Sequence Diagram]



Save Address Book Use Case

The Save Address Book use case is initiated when the user chooses "Save" from the File menu. (The Save option is grayed out unless changes have been made to the address book since the last New, Open, Save, or Save As ... operation was done.) If there is a current file, the current address book is saved to this file. (If attempting to write the file results in an error, a dialog box is displayed warning the user of the error.) If there is no current file, the Save Address Book As .. use case is done instead. In all cases, the current address book and window list are left unchanged.

[Sequence Diagram]



Save Address Book As ... Use Case

The Save Address Book As ... use case is initiated when the user chooses "Save As ..." from the File menu. (The Save As ... option is always available.) A save file dialog is displayed and the user is allowed to choose the name of a file in which to save the address book. (If the user cancels the file dialog, the Save As ... operation is canceled.) The current address book is saved to the specified file, and the file to which it was saved becomes the current file and its name is displayed as the title of the main window. (If attempting to write the file results in an error, a dialog box is displayed warning the user of the error, and the current file and main window title are unchanged.) In all cases, the current address book and window list are left unchanged.

[Sequence Diagram]

Quit Program Use Case

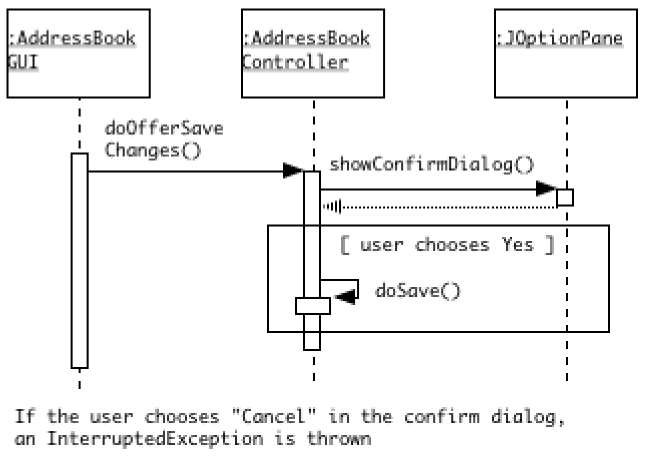
The Quit Program use case is initiated when the user chooses "Quit" from the File menu, or clicks the close box for the main window. In either case, if the current address book contents have been changed since the last New, Open, Save, or Save As ... operation was done, the Offer to Save Changes extension is executed. Unless the user cancels the operation, the program is terminated.

[Sequence Diagram]

Offer to Save Changes Extension

The Offer to Save Changes extension is initiated from within the Create New Address Book, Open Existing Address Book, or Quit program use cases, if the current address book has been changed since the last successful New, Open, Save, or Save As ... operation was done. A dialog box is displayed, informing the user that there are unsaved changes, and asking the user whether to save changes, not save changes, or cancel the operation. If the user chooses to save changes, the Save Address Book Use Case is executed (which may result in executing the Save Address Book As ... Use Case if there is no current file). If the user chooses not to save changes, the original operation is simply resumed. If the user chooses to cancel (or cancels the save file dialog if one is needed), the original operation is canceled.

[Sequence Diagram]

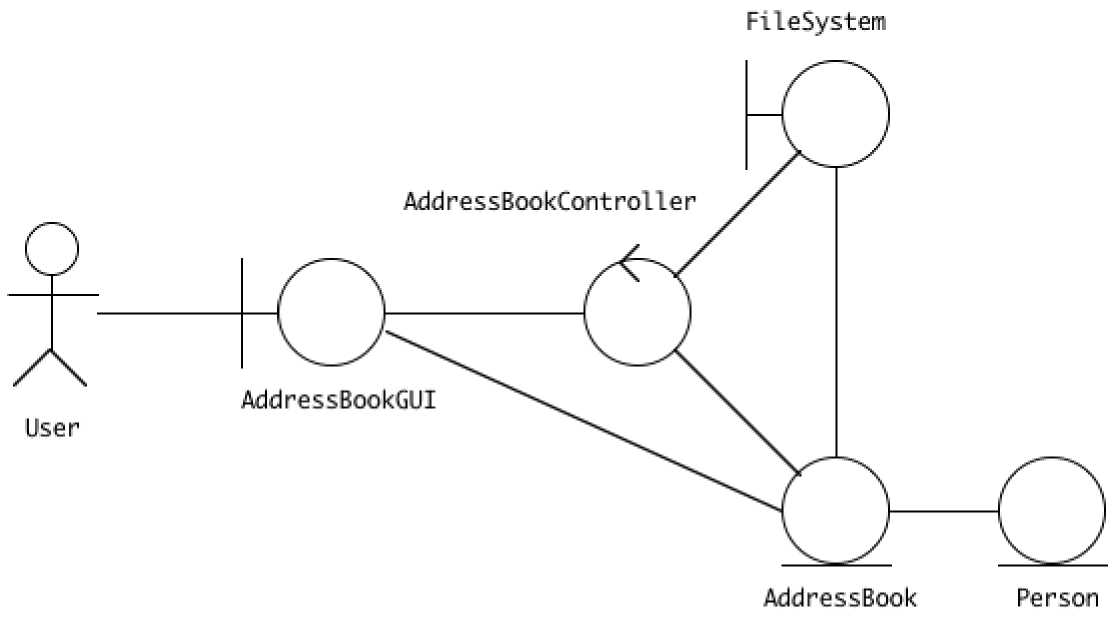


Analysis

An initial reading of the use cases suggests that the following will be part of the system.

- A single entity object representing the current address book that the program is working with (`AddressBook`).
- An arbitrary number of entity objects, each representing one of the people that is in the current address book (`Person`).
- A boundary object representing the interface between the address book system and the human user (`AddressBookGUI`).
- A boundary object representing the interface between the address book system and the file system on disk (`FileSystem`).

- A controller object that carries out the use cases in response to user gestures on the GUI (`AddressBookController`). (For a problem of this small size, a single controller is sufficient.)



The various use cases work with these objects, as follows:

- The Add a Person Use Case involves getting the new information from the user, and then telling the AddressBook object to add a new person with this information to its collection
- The Edit a Person Use Case involves displaying the current information about the desired person (obtained from the AddressBook), then allowing the user to enter new information for the various fields, then telling the AddressBook object to make the changes.
- The Delete a Person Use Case involves asking the user to confirm deletion, and then telling the AddressBook object to remove this person from its collection.
- The Sort Entries by Name Use Case involves telling the AddressBook object to rearrange its collection in order of name.
- The Sort Entries by ZIP Use Case involves telling the AddressBook object to rearrange its collection in order of ZIP.
- The Create New Address Book Use Case involves creating a new AddressBook object.
- The Open Existing Address Book Use Case involves getting a file specification from the user, and then telling the FileSystem object to read in an AddressBook object from this file.

- The Save Address Book Use Case involves determining whether or not the current AddressBook object has a file it was last read from / saved to; if so, telling the FileSystem object to save the current AddressBook object to this file. (If not, the Save Address Book As ... Use Case is done instead.)
- The Save Address Book As ... Use Case involves getting a file specification from the user, and then telling the FileSystem object to save the current AddressBook object to this file.
- The Print Address Book Use Case involves telling the AddressBook object to print out its collection in order.
- (The Quit Program Use Case does not involve any of the other objects)
- (The Offer to Save Changes Extension may involve performing the Save Address Book Use Case.)

CRC Cards for the Address Book Example

Responsibilities are assigned to the various classes based on the use of the model-view-controller design pattern. The two entity classes (AddressBook and Person) serve as the model. The GUI class (AddressBookGUI) serves as the view. The controller class (AddressBookController) serves, of course, as the controller.

The view (AddressBookGUI) needs to be made an observer of the model (specifically, AddressBook) so that it always reflects the current state of the model - specifically, the list of names, the title, and its saved/needs to be saved status.

Using CRC cards to assign responsibilities to various classes for the tasks required by the various use cases leads to the creation of the following cards.

- Class AddressBook
- Class AddressBookController
- Class AddressBookGUI
- Class FileSystem
- Class Person

Class AddressBook

The CRC Cards for class AddressBook are left as an exercise to the student

[Links for this class]

Class AddressBookController

The basic responsibility of an AddressBookController object is to carry out the various use cases.

Responsibilities	Collaborators
Allow the user to perform the Add a Person Use Case	AddressBook
Allow the user to perform the Edit a Person Use Case	AddressBook
Allow the user to perform the Delete a Person Use Case	AddressBook
Allow the user to perform the Sort Entries by Name Use Case	AddressBook
Allow the user to perform the Sort Entries by ZIP Use Case	AddressBook
Allow the user to perform the Create New Address Book Use Case	AddressBook
Allow the user to perform the Open Existing Address Book Use Case	FileSystem
Allow the user to perform the Save Address Book Use Case	AddressBook FileSystem
Allow the user to perform the Save Address Book As ... Use Case	FileSystem
Allow the user to perform the Print Entries Use Case	AddressBook
Perform the Offer to Save Changes Extension when needed by another Use Case	AddressBook

[Links for this class]

Class AddressBookGUI

The basic responsibility of a GUI object is to allow interaction between the program and the human user.

Responsibilities	Collaborators
Keep track of the address book object it is displaying	
Display a list of the names of persons in the current address book	AddressBook
Display the title of the current address book - if any	AddressBook
Maintain the state of the "Save" menu option - usable only when the address book has been changed since the last time it was opened / saved.	AddressBook
Allow the user to request the performance of a use case	AddressBookController

[Links for this class]

Class FileSystem

The basic responsibility of a FileSystem object is to manage interaction between the program and the file system of the computer it is running on.

Responsibilities	Collaborators

Read a stored address book from a file, given its file name	AddressBook
Save an address book to a file, given its file name	AddressBook

[Links for this class]

Class Person

The basic responsibility of a Person object is to maintain information about a single individual.

Responsibilities	Collaborators
Create a new object, given an individual's name, address, city, state, ZIP, and phone	
Furnish the individual's first name	
Furnish the individual's last name	
Furnish the individual's address	
Furnish the individual's city	
Furnish the individual's state	
Furnish the individual's ZIP	
Furnish the individual's phone number	
Update the stored information (except the name) about an individual	

[Links for this class]

Class Diagram for the Address Book Example

Shown below is the class diagram for the Address Book Example. To prevent the diagram from becoming overly large, only the name of each class is shown - the attribute and behavior "compartments" are shown in the detailed design, but are omitted here.

The diagram includes the classes discovered during analysis, plus some additional classes discovered during design. (In a more significant system, the total number of classes may be about five times as great as the number of classes uncovered during analysis.)

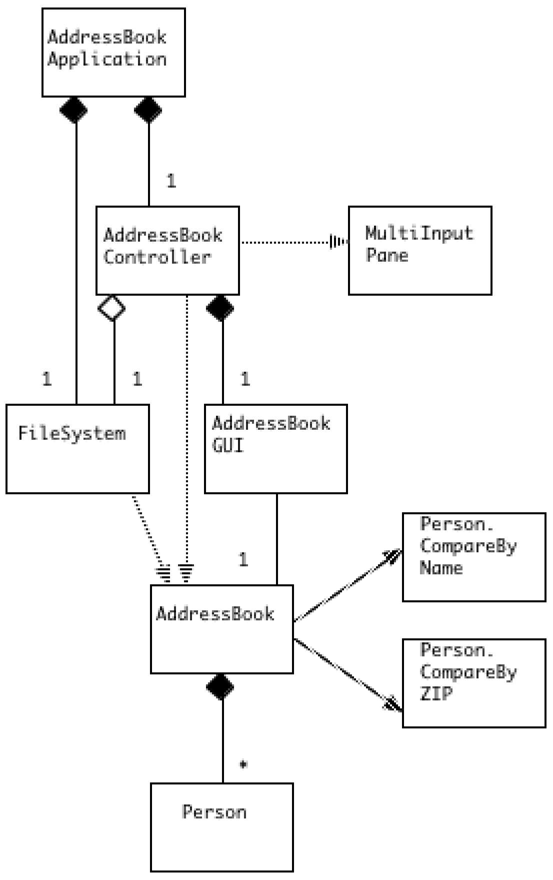
- AddressBookApplication - main class for the application; responsible for creating the FileSystem and GUI objects and starting up the application.
- MultiInputPane - a utility class for reading multiple values at a single time. (Design not further documented, but javadoc is included.)
- Person.CompareByName - Comparator for comparing two Person objects by name (used for sorting by name).

- Person.CompareByZip - Comparator for comparing two Person objects by zip (used for sorting by name).

The following relationships hold between the objects:

- The main application object is responsible for creating a single file system object and a single controller object.
- The file system object is responsible for saving and re-loading address books
- The controller object is responsible for creating a single GUI object.
- The controller object is responsible for initially creating an address book object, but the GUI is henceforth responsible for keeping track of its current address book - of which it only has one at any time.
- The GUI object and the address object are related by an observer-observable relationship, so that changes to the address book content lead to corresponding changes in the display
- The address book object is responsible for creating and keeping track of person objects, of which there can be many in any given address book.
- A MultiInputPane object is used by the controller to allow the user to enter multiple items of data about a person.
- A comparator object of the appropriate kind is used by the address book object when sorting itself.

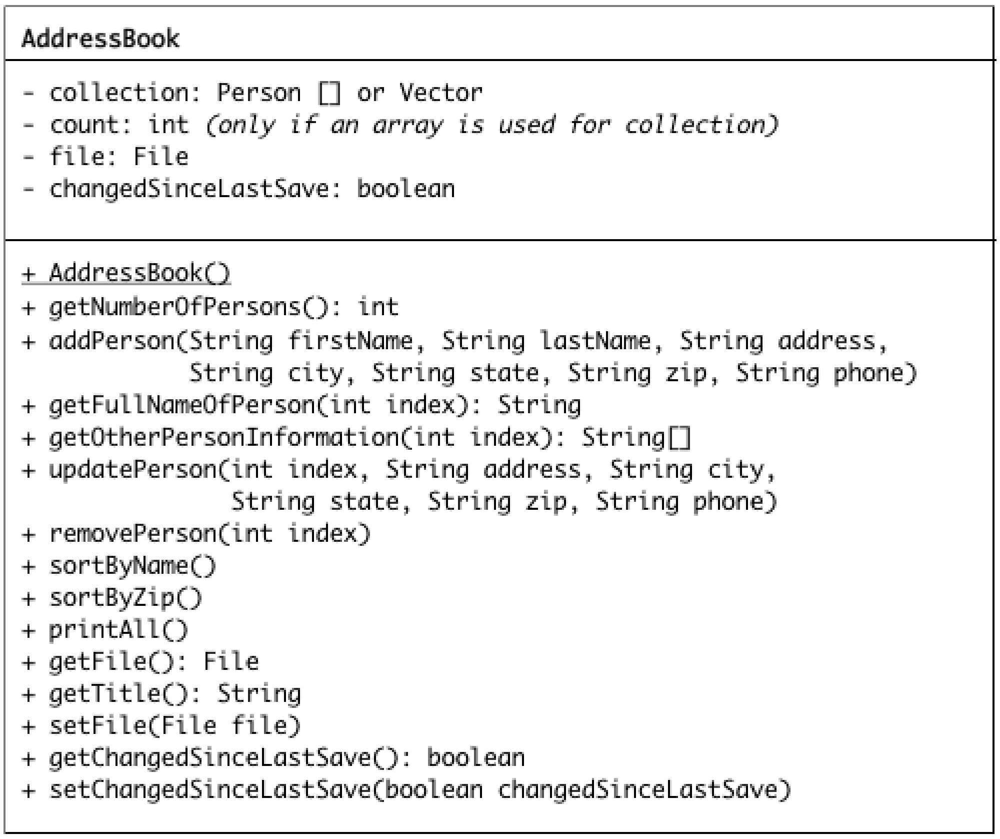
Click on a class icon for links to further information about it



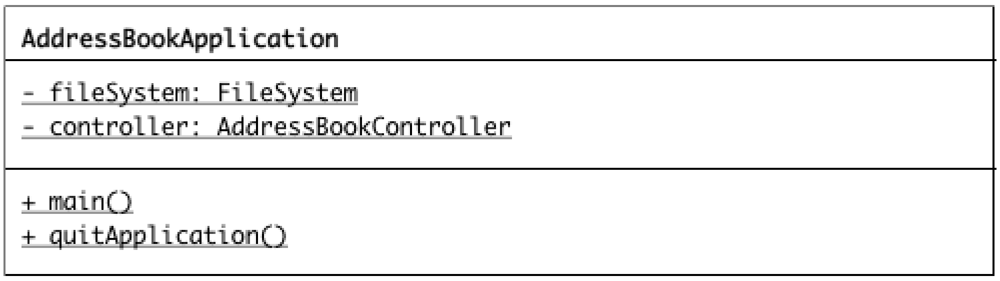
Detailed Class Design for the Address Book Example

Given below is a "three compartment" design for the classes appearing in the class diagram. This information was not included in that diagram due to size considerations; however, it could have been - in which case this document would have been unnecessary.

- Class AddressBook
- Class AddressBookApplication
- Class AddressBookController
- Class AddressBookGUI
- Class FileSystem
- Class Person
- (No detailed design is given for Comparator class Person.CompareByName)
- (No detailed design is given for Comparator class Person.CompareByZip)
- (No detailed design is given for utility class MultiInputPane)



```
AddressBook
---------------------------------------------------------------
- collection: Person [] or Vector
- count: int (only if an array is used for collection)
- file: File
- changedSinceLastSave: boolean
---------------------------------------------------------------
+ AddressBook()
+ getNumberOfPersons(): int
+ addPerson(String firstName, String lastName, String address,
            String city, String state, String zip, String phone)
+ getFullNameOfPerson(int index): String
+ getOtherPersonInformation(int index): String[]
+ updatePerson(int index, String address, String city,
               String state, String zip, String phone)
+ removePerson(int index)
+ sortByName()
+ sortByZip()
+ printAll()
+ getFile(): File
+ getTitle(): String
+ setFile(File file)
+ getChangedSinceLastSave(): boolean
+ setChangedSinceLastSave(boolean changedSinceLastSave)
```



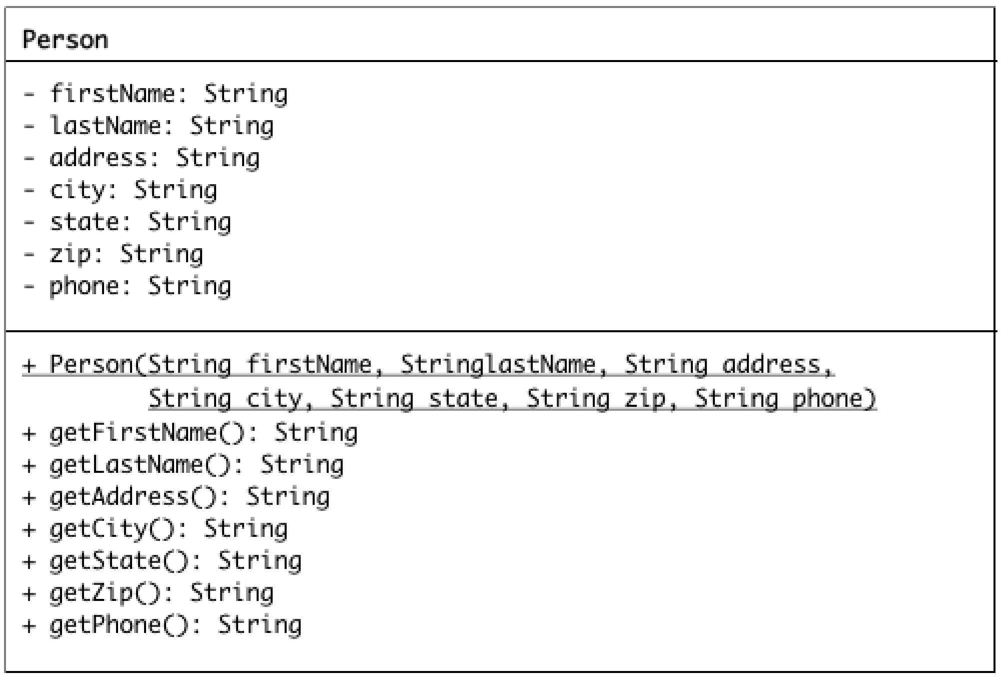
```
AddressBookApplication
---------------------------------------------------------------
- fileSystem: FileSystem
- controller: AddressBookController
---------------------------------------------------------------
+ main()
+ quitApplication()
```

The detailed design of class AddressBookController is left as an exercise to the student

```
AddressBookGUI
--------------------------------------------------
- controller: AddressBookController
- addressBook: AddressBook
- nameListModel: AbstractListModel
- nameList: JList
- addButton: JButton
- editButton: JButton
- deleteButton: JButton
- sortByNameButton: JButton
- sortByZipButton: JButton
- newItem: JMenuItem
- openItem: JMenuItem
- saveItem: JMenuItem
- saveAsItem: JMenuItem
- printItem: JMenuItem
- quitItem: JMenuItem
--------------------------------------------------
+ AddressBookGUI(AddressBookController controller,
     AddressBook addressBook)
+ getAddressBook(): AddressBook
+ setAddressBook(AddressBook addressBook)
+ reportError(String message)
+ update(Observable o, Object arg)
```

```
FileSystem
--------------------------------------------------

--------------------------------------------------
+ readFile(File file): AddressBook
+ saveFile(AddressBook addressBook, File file)
```



Code for Simple Address Book Example

As noted in the introduction, the writing of much of the code for this problem is an assignment in two closed labs and an open lab programming project in one of the courses I teach. This page includes links to portions of the code that are not assigned (and are, in fact, given to the students in the course.)

This page also provides access to Complete Javadoc Documentation for all of the classes.

- AddressBookApplication
- AddressBookGUI
- MultiInputPane
- Comparator Classes - inner classes in class Person

AddressBookApplication

```
/*
 * AddressBookApplication.java
 *
 * Main program for address book application
 *
 * Copyright (c) 2001, 2003, 2005 - Russell C. Bjork
 *
 */

import java.awt.Frame;
import java.awt.event.WindowEvent;
```

```
// The next line is only needed on the Mac platform - comment out
// if compiling on some other platform (but comment back in and recompile
// before moving final version to server)

import com.apple.eawt.*;

/** Main class for the Address Book example
 */
public class AddressBookApplication
{
    /** Main method for program
     */
    public static void main(String [] args)
    {
                FileSystem fileSystem = new FileSystem();
                AddressBookController controller = new
AddressBookController(fileSystem);
                AddressBookGUI gui = new AddressBookGUI(controller, new
AddressBook());
                gui.show();

                // Register a Mac quit handler - comment out if compiling
on some
                // other platform (but comment back in and recompile
                // before moving final version to server)

                com.apple.eawt.Application application =
                        com.apple.eawt.Application.getApplication();
                application.addApplicationListener(new ApplicationAdapter()
{
                        public void handleQuit(ApplicationEvent e)
                        {
                                e.setHandled(false);
                                quitApplication();
                        }
                });
    }

    /** Terminate the application (unless cancelled by the user)
     */
    public static void quitApplication()
    {
                // When the user requests to quit the application, any open
                // windows must be closed
                Frame [] openWindows = Frame.getFrames();
                for (int i = 0; i < openWindows.length; i ++)
                {
                        // Attempt to close any window that belongs to this
program

                        if (openWindows[i] instanceof AddressBookGUI)
                        {
                                openWindows[i].dispatchEvent(new
WindowEvent(

openWindows[i],

WindowEvent.WINDOW_CLOSING));

                 // If the window is still showing, this means that this
attempt
```

```
                 // to close the window was cancelled by the user - so abort
the
                 // quit operation

                                 if (openWindows[i].isShowing())
                     return;
                        }
                }

                // If we get here, all open windows have been successfully
closed
                // (i.e. the user has not cancelled an offer to save any of
them).
                // Thus, the application can terminate.

                System.exit(0);
        }
}
```

AddressBookGUI

```
/**
 *      AddressBookGUI.java
 *
 *      Copyright (c) 2000, 2001, 2005 - Russell C. Bjork
 *
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.IOException;
import java.util.Observer;
import java.util.Observable;

/**     An object of this class allows interaction between the program and
the
 *      human user.
 */
public class AddressBookGUI extends JFrame implements Observer
{
    /** Constructor
     *
     *  @param controller the controller which performs operations in
     *              response to user gestures on this GUI
     *  @param addressBook the AddressBook this GUI displays
     */

    public AddressBookGUI(final AddressBookController controller,
                                        AddressBook addressBook)
    {
                this.controller = controller;

           // Create and add file menu

          JMenuBar menuBar = new JMenuBar();
          JMenu fileMenu = new JMenu("File");
          newItem = new JMenuItem("New", 'N');
          fileMenu.add(newItem);
          openItem = new JMenuItem("Open...", 'O');
          fileMenu.add(openItem);
          fileMenu.addSeparator();
          saveItem = new JMenuItem("Save", 'S');
          fileMenu.add(saveItem);
          saveAsItem = new JMenuItem("Save As...");
          fileMenu.add(saveAsItem);
          fileMenu.addSeparator();
          printItem = new JMenuItem("Print", 'P');
          fileMenu.add(printItem);
          fileMenu.addSeparator();
          quitItem = new JMenuItem("Quit", 'Q');
          fileMenu.add(quitItem);
          menuBar.add(fileMenu);
          setJMenuBar(menuBar);

                // The displayed list of names gets its information from
the
                // address book

                nameListModel = new NameListModel();

                // The nameListModel and saveItem objects must exist before
this is done;
                // but this must be done before the nameList is created

                setAddressBook(addressBook);

         // Create and add components for the main window

          nameList = new JList(nameListModel);
          JScrollPane listPane = new JScrollPane(nameList);
          nameList.setVisibleRowCount(10);
          listPane.setBorder(BorderFactory.createCompoundBorder(
              BorderFactory.createEmptyBorder(10, 10, 10, 10),
              BorderFactory.createLineBorder(Color.gray, 1)));
          getContentPane().add(listPane, BorderLayout.CENTER);
          JPanel buttonPanel = new JPanel();
          addButton = new JButton("     Add    ");
          buttonPanel.add(addButton);
          editButton = new JButton("    Edit    ");
          buttonPanel.add(editButton);
          deleteButton = new JButton("   Delete   ");
          buttonPanel.add(deleteButton);
          sortByNameButton = new JButton("Sort by name");
          buttonPanel.add(sortByNameButton);
          sortByZipButton = new JButton("Sort by ZIP ");
          buttonPanel.add(sortByZipButton);
          buttonPanel.setBorder(BorderFactory.createEmptyBorder(5, 10, 10,
10));

          getContentPane().add(buttonPanel, BorderLayout.SOUTH);
```

```
        // Add the action listeners for the buttons, menu items, and close
box,
        // and for double-clicking the list

        addButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
               controller.doAdd(AddressBookGUI.this);
               int index = getAddressBook().getNumberOfPersons() - 1;
               // This will ensure that the person just added is visible
in list
               nameList.ensureIndexIsVisible(index);
            }
        });

        editButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                              int index = nameList.getSelectedIndex();
                              if (index < 0)
                                      reportError("You must select a
person");
                              else

        controller.doEdit(AddressBookGUI.this, index);
            }
        });

        deleteButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
               int index = nameList.getSelectedIndex();
                              if (index < 0)
                                      reportError("You must select a
person");
                              else
                      controller.doDelete(AddressBookGUI.this, index);
            }
        });

        sortByNameButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
               controller.doSortByName(AddressBookGUI.this);
             }
        });

        sortByZipButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                              controller.doSortByZip(AddressBookGUI.this);
            }
        });

        newItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                              try
                              {
```

```
                                        if
(getAddressBook().getChangedSinceLastSave())

        controller.doOfferSaveChanges(AddressBookGUI.this);


        controller.doNew(AddressBookGUI.this);
                                }
                catch(IOException exception)
                {
                        reportError("Problem writing the file: " +
                                        exception);
                }
                catch(InterruptedException exception)
                {
                        // Thrown if user cancels a save or a file dialog -
can be ignored
                }
                                catch(SecurityException exception)
                                {
                                        // Thrown if security manager
disallows the operation -
                                        // will always happen in an applet

                                        reportError("Operation disallowed: "
+ exception);
                                }
            }
        });

        openItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                                try
                                {
                                        if
(getAddressBook().getChangedSinceLastSave())

        controller.doOfferSaveChanges(AddressBookGUI.this);

                        controller.doOpen(AddressBookGUI.this);
                }
                catch(IOException exception)
                {
                        reportError("Problem reading or writing the file: "
+
                                        exception);
                }
                catch(InterruptedException exception)
                {
                        // Thrown if user cancels a save or a file dialog -
can be ignored
                }
                                catch(SecurityException exception)
                                {
                                        // Thrown if security manager
disallows the operation -
                                        // will always happen in an applet

                                        reportError("Operation disallowed: "
+ exception);
```

```
                               }
                catch(Exception exception)
                {
                        // Any other case means the file did not contain an
                        // address book

                        reportError("This file did not contain an address
book");
                }
            }
        });

       saveItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                try
                {
                        controller.doSave(AddressBookGUI.this);
                }
                catch(IOException exception)
                {
                        reportError("Problem writing the file: " +
                                                exception);
                }
                catch(InterruptedException exception)
                {
                        // Thrown if user cancels a file dialog - can be
ignored
                }
                               catch(SecurityException exception)
                               {
                                       // Thrown if security manager
disallows the operation -
                                       // will always happen in an applet

                                       reportError("Operation disallowed: "
+ exception);
                               }
            }
        });

        saveAsItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                try
                {
                        controller.doSaveAs(AddressBookGUI.this);
                }
                catch(IOException exception)
                {
                        reportError("Problem writing the file: " +
                                                exception);
                }
                catch(InterruptedException exception)
                {
                        // Thrown if user cancels a file dialog - can be
ignored
                }
                               catch(SecurityException exception)
                               {
                                       // Thrown if security manager
disallows the operation -
                                       // will always happen in an applet

                                       reportError("Operation disallowed: "
+ exception);
                               }
            }
        });

        printItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                               controller.doPrint(AddressBookGUI.this);
                       }
        });

        quitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                               AddressBookApplication.quitApplication();
                       }
        });

               setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e)
            {
                               try
                               {
                                       if
(getAddressBook().getChangedSinceLastSave())

        controller.doOfferSaveChanges(AddressBookGUI.this);

                                       dispose();

                                       if (Frame.getFrames().length == 0)

        AddressBookApplication.quitApplication();
                               }
               catch(IOException exception)
               {
                       reportError("Problem writing the file: " +
                                               exception);
               }
               catch(InterruptedException exception)
               {
                       // Thrown if user cancels a file dialog - can be
ignored
               }
                               catch(SecurityException exception)
                               {
                                       // Thrown if security manager
disallows the operation -
                                       // will always happen in an applet

                                       reportError("Operation disallowed: "
+ exception);
                               }
            }
        });

               // The following is adapted from an example in the
documentation
               // for class JList.  It invokes the controller's doEdit
method
               // if the user double clicks a name.

               nameList.addMouseListener(new MouseAdapter() {
                       public void mouseClicked(MouseEvent e)
                       {
                               if (e.getClickCount() == 2)
                               {
                                       int index =
nameList.locationToIndex(e.getPoint());

        controller.doEdit(AddressBookGUI.this, index);
                               }
                       }
               });

               pack();
   }

        /** Accessor for the address book this GUI displays
         *
         *      @return the current address book for this GUI
         */
        public AddressBook getAddressBook()
        {
               return addressBook;
        }

        /** Mutator to change the address book this GUI displays
         *
         *      @param addressBook the new address book for this GUI
         */
        public void setAddressBook(AddressBook addressBook)
        {
               if (this.addressBook != null)
                       this.addressBook.deleteObserver(this);

               this.addressBook = addressBook;
               addressBook.addObserver(this);
               update(addressBook, null);
        }

   /** Report an error to the user
    *
    *  @param message the message to display
    */
   public void reportError(String message)
   {
        JOptionPane.showMessageDialog(this, message, "Error message",
                                       JOptionPane.ERROR_MESSAGE);
   }

        /** Method required by the Observer interface - update the display
         *      in response to any change in the address book
         */
        public void update(Observable o, Object arg)
        {
               if (o == addressBook)
               {
                       setTitle(addressBook.getTitle());

        saveItem.setEnabled(addressBook.getChangedSinceLastSave());
                       nameListModel.contentsChanged();
               }
        }

   // GUI components and menu items

        private NameListModel nameListModel;
   private JList nameList;
   private JButton addButton, editButton, deleteButton, sortByNameButton,
sortByZipButton;
   private JMenuItem newItem, openItem, saveItem, saveAsItem, printItem,
quitItem;

        // The controller that performs operations in response to user
gestures

        private AddressBookController controller;

        // The address book this GUI displays / operates on

        private AddressBook addressBook;

        /** Class used for the model for the list of persons in the address
book
         */
        private class NameListModel extends AbstractListModel
        {
               /** Report that the contents of the list have changed
                */
               void contentsChanged()
               {
                       super.fireContentsChanged(this, 0, 0);
               }

               // Implementation of abstract methods of the base class

               public Object getElementAt(int index)
               {
                       return getAddressBook().getFullNameOfPerson(index);
               }

               public int getSize()
               {
                       return getAddressBook().getNumberOfPersons();
               }
        }
}
```

MultiInputPane

```
/*
 * MultiInputPane.java
 *
 * Copyright (c) 2004, 2005 - Russell C. Bjork
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/** This is a utility class for displaying a dialog that asks for multiple
values.
 *      Based on ideas in Wu's javabook.MultiInputBox class and on ideas in
 *      javax.swing.JOptionPane
 */
public class MultiInputPane extends JOptionPane
{
        /** Pop up a dialog asking for multiple items of input
         *
         *      @param parentComponent the parent Component of the dialog
that is shown
         *      @param prompts the prompts to display
         *      @param initialValues the initial values to display for each
item -
         *                this parameter can be null, in which case no
initial values
         *                are specified; or individual elements can be
null,
         *                indicating that no initial value is specified for
a particular
         *                field
         *      @param title the title for this dialog
         *      @return an array of values corresponding to the various
prompts,
         *                      or null if the user cancelled
         */
        public static String [] showMultiInputDialog(Component
parentComponent,

                        String [] prompts,

                        String [] initialValues,

                        String title)
        {
                MultiInputPane pane = new MultiInputPane(prompts,
initialValues);
                JDialog dialog = pane.createDialog(parentComponent,

title != null ? title : "MultiInputPane");
                dialog.pack();
                dialog.show();

                if (! pane.ok)
                        return null;

                String [] results = new String [prompts.length];
                for (int i = 0; i < prompts.length; i ++)
                        results[i] = pane.fields[i].getText();
                return results;
        }

        /** Pop up a dialog asking for multiple items of input
         *
         *      @param parentComponent the parent Component of the dialog
that is shown
         *      @param prompts the prompts to display
         *      @param title the title for this dialog
         *      @return an array of values corresponding to the various
prompts,
         *                      or null if the user cancelled
         */
        public static String [] showMultiInputDialog(Component
parentComponent,

                        String [] prompts,

                        String title)
        {
                return showMultiInputDialog(parentComponent, prompts, null,
title);
        }

        /** Pop up a dialog asking for multiple items of input
         *
         *      @param parentComponent the parent Component of the dialog
that is shown
         *      @param prompts the prompts to display
         *      @param initialValues the initial values to display for each
item -
         *                this parameter can be null, in which case no
initial values
         *                are specified; or individual elements can be
null,
         *                indicating that no initial value is specified for
a particular
         *                field
         *      @return an array of values corresponding to the various
prompts,
         *                      or null if the user cancelled
         */
        public static String [] showMultiInputDialog(Component
parentComponent,

                        String [] prompts,

                        String [] initialValues)
        {
                return showMultiInputDialog(parentComponent, prompts,
initialValues, null);
        }

        /** Pop up a dialog asking for multiple items of input
         *
         *      @param parentComponent the parent Component of the dialog
that is shown
         *      @param prompts the prompts to display
         *      @return an array of values corresponding to the various
prompts,
         *                      or null if the user cancelled
         */
        public static String [] showMultiInputDialog(Component
parentComponent,

                        String [] prompts)
        {
                return showMultiInputDialog(parentComponent, prompts, null,
null);
        }

        /** Constructor used by the above
         *
         *      @param prompts the prompts to display
         *      @param initialValues the initial values to display for each
item -
         *                this parameter can be null, in which case no
initial values
         *                are specified; or individual elements can be
null,
         *                indicating that no initial value is specified for
a particular
         *                field
         */
        private MultiInputPane(String [] prompts, String [] initialValues)
        {
                super();
                removeAll();

                setLayout(new GridLayout(prompts.length + 1, 2, 5, 5));
                fields = new JTextField[prompts.length];

                for (int i = 0; i < prompts.length; i ++)
                {
                        add(new JLabel(prompts[i]));
                        fields[i] = new JTextField();
                        add(fields[i]);
                        if (initialValues != null && initialValues[i] !=
null)
                                fields[i].setText(initialValues[i]);
                }

                JPanel okPanel = new JPanel();
                JButton okButton = new JButton("OK");
                okPanel.add(okButton);
                add(okPanel);
                JPanel cancelPanel = new JPanel();
                JButton cancelButton = new JButton("Cancel");
                cancelPanel.add(cancelButton);
                add(cancelPanel);

                okButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e)
                        {
                                ok = true;
                                getTopLevelAncestor().setVisible(false);
                        }
                });

                cancelButton.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent e)
                        {
                                getTopLevelAncestor().setVisible(false);
                        }
                });
        }

        private JTextField [] fields;
        private boolean ok;
}
```

Comparator Classes - inner classes in class Person

```
// The following comparators should go inside class Person:


        /** Comparator for comparing two persons by alphabetical order of
name
         */
        public static class CompareByName implements Comparator
        {
                /** Compare two objects (which must both be Persons) by
last name,
                 *      with ties broken by first name
                 *
                 *      @param person1 the first object
                 *      @param person2 the second object
                 *      @return a negative number if person1 belongs before
person2 in
                 *                      alphabetical order of name; 0 if they
are equal; a
                 *                      positive number if person1 belongs
after person2
                 *
                 *      @exception ClassCastException if either parameter is
not a
                 *                              Person object
                 */
                public int compare(Object person1, Object person2)
                {
                        int compareByLast = ((Person)
person1).getLastName().compareTo(
                                ((Person) person2).getLastName());
                        if (compareByLast != 0)
                                return compareByLast;
                        else
                                return ((Person)
person1).getFirstName().compareTo(
                                        ((Person) person2).getFirstName());
                }

                /** Compare two objects (which must both be Persons) by
name
                 *
                 *      @param person1 the first object
                 *      @param person2 the second object
                 *      @return true if they have the same name, false if
they do not
                 *
                 *      @exception ClassCastException if either parameter is
not a
                 *                          Person object
                 */
                public boolean equals(Object person1, Object person2)
                {
                        return compare(person1, person2) == 0;
                }
        }

        /** Comparator for comparing two persons by order of zip code
         */
        public static class CompareByZip implements Comparator
        {
                /** Compare two objects (which must both be Persons) by zip
                 *
                 *      @param person1 the first object
                 *      @param person2 the second object
                 *      @return a negative number if person1 belongs before
person2 in
                 *                      order of zip; 0 if they are equal; a
positive number if
                 *                  person1 belongs after person2
                 *
                 *      @exception ClassCastException if either parameter is
not a
                 *                          Person object
                 */
                public int compare(Object person1, Object person2)
                {
                        int compareByZip = ((Person)
person1).getZip().compareTo(
                                ((Person) person2).getZip());
                        if (compareByZip != 0)
                                return compareByZip;
                        else
                                return new CompareByName().compare(person1,
person2);
                }

                /** Compare two objects (which must both be Persons) by zip
                 *
                 *      @param person1 the first object
                 *      @param person2 the second object
                 *      @return true if they have the same zip, false if
they do not
                 *
                 *      @exception ClassCastException if either parameter is
not a
                 *                          Person object
                 */
                public boolean equals(Object person1, Object person2)
                {
                        return compare(person1, person2) == 0;
                }
        }
```

Maintenance

This page lists various changes that might be made to the system. Modifying the various documents to incorporate one or more of these changes would make an interesting exercise for the reader. They are listed in order of estimated increasing difficulty.

- The Print Entries Use Case currently sends printed output to System.out. Alternately, it could send its output to a file, chosen by the user in response to a file dialog.
- It was noted in the original requirements that the program might be modified to allow multiple address books to be open at the same time - each in its own window. This might entail the following changes:
    - The Create New Address Book and Open Existing Address Book Use Cases would no longer close the current address book. Instead, they would create a new copy of the GUI, with its own address book (either a newly created, empty one, or one read from a file specified by the user.) There would thus be two (or more) windows visible on the screen.

    - A new Close Address Book Use Case would be added to allow the user to close a single window (and its associated address book). This could be initiated by a new Close option in the File menu, or by clicking the close box for the window. It would offer to save changes, if necessary, and then close the window. If the window that is closed is the last open window in the program, then the program should be terminated as well; otherwise, the program would continue running with the remaining window(s) visible.
    - The code that is activated when the close box for the window is clicked would be the Close Address Book Use Case described above, instead of the Quit Program Use Case.
    - The Quit Program Use Case (activated from the Quit item in the File menu) would need to cause all open windows to be closed, with appropriate offers to save changes, unless the user cancels the operation for any window. If the user cancels the save for any window, the entire use case would terminate at once, without attempting to close additional windows.

- A facility might be created that would allow the user to search for the occurrence of some character string anywhere in the information about a person. For example, searching for the string "Buffalo" might find Boris Buffalo or a person living in Buffalo, NY; searching for the string "0191" might find a person living in ZIP code 01915 or a person whose phone number is 555-0191, etc. This might entail adding two new use cases: Find and Find Again.

    - The Find Use Case could pop up a dialog asking the user to enter a character string, and would then search through all the people in the address book (starting at the beginning) until a person is found for which the string occurs anywhere in the stored information (in either name, in the address, etc.) This person would then be selected in the displayed list of names.
    - The Find Again Use Case could look for the next occurrence of the same string, beginning where the previous Find/Find Again left off.

      Of course, this option would not be available when a search is not in progress - e.g. when an address book is newly created or opened, or when the previous Find/Find Again did not find anyone. It would also be reasonable to disable this option when any change is made to the address book (e.g. by Add, Edit, Delete, or a Sort).

  To allow these two new use cases to be initiated, a new Search menu could be added with two choices, perhaps labelled Find and Find Again. In this case, Find Again would be grayed out when the Find Again Use Case is not available; and Find might also be grayed out when the address book is totally empty.

Some good practice in working with UML might come by modifying the various design documents (beginning with the use cases), not just changing the code.