CSC 309 Sample Final Exam

Given below are excerpts from the requirements specification for a form letter mailing system. This is the same system as on the midterm. Compared to the midterm, the requirements specification on the final has some additional scenarios and additions to the formal specification. You need not read all of the requirements specification in advance; refer back to it as necessary to answer the specific exam questions.

Following the requirements specification are the final exam questions. The exam is open-book, open-note. It is worth a total of 180 points. You have three hours (180 minutes) to complete it; figure roughly one minute per point.

1. Introduction

Acme Junkmail Inc. needs to make their business more efficient. At present, they have fifty employees who produce individual advertising letters using a standard word processor. The letters are mailed to potential customers who may be interested in purchasing certain products. Acme wants a more automated form of letter generation system.

2. Functional Requirements

The Acme system maintains three databases:

1. A form letter database, consisting of form letter templates.
2. A prize database, consisting of prizes that can be offered to customers
3. A customer database, consisting of potential customers to whom letters will be mailed.

For database management, the system provides operations to add, delete, modify, and search for records in any of the three databases. To produce form letters, the system provides a text editor for creating form letter templates. A template consists of standard text and template fill-in fields. For example, a typical form letter could have fill-in fields for customer name, address, and other specialized information.

To generate customized letters, the user selects a template from the form letter database, and fills in selection criteria for which customers will receive the letter. For example, selection criteria could be all customers in a particular city between the ages of 20 and 50. After selection criteria are defined, letters are generated by finding all customer records that match the criteria. For each matching customer, a personalized letter is generated by filling in the template fill-in fields with specific information from the customer database record.

2.1. User Interface Overview

Figure 1 shows an expansion of the Acme system command menus.

The `File` and `Edit` menus have typical commands for manipulating data files and basic text editing.

The `Customers` and `Prizes` menus provide standard commands to manage each of the databases. The `Add` command adds a new record. The `Delete` command deletes an existing record. The `Change` command changes an existing record. The `Find` command finds one or more records by customer name.

The `Letters` menu provides operations to create, delete, edit, and find a form letter template. The '`Generate ...`' command allows the user to perform the generation of form letters based on selection criteria.

The `Options` menu allows the user to set options for system operation.

2.2. Customer Database Management

When the user selects the '`Add ...`' item from the `Customers` menu, the system displays the dialog shown in Figure 2. The user enters free-form string values in the `Name`, `Company`, `Age`, and `Address` fields. The `Field Name` and `Field Value` columns are typeable text areas in which the user enters customer-specific data fields.

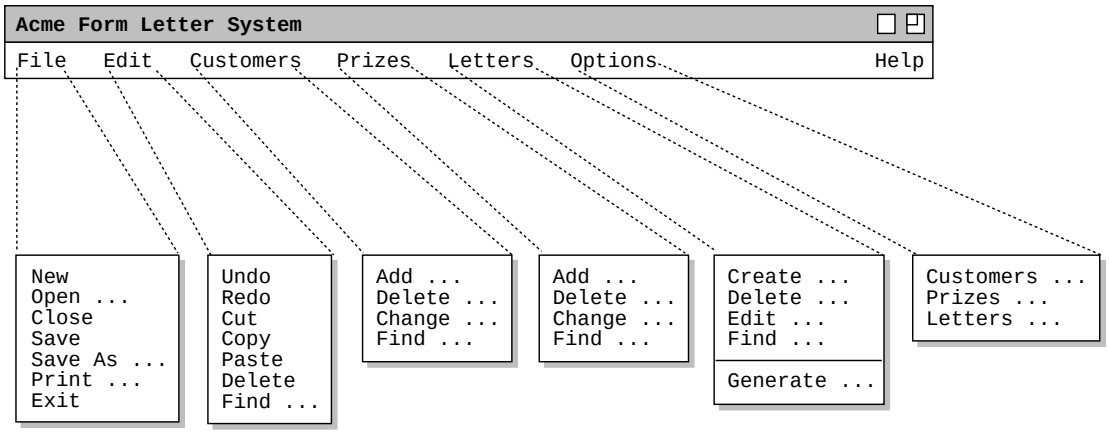


Figure 1: Expanded command menus.

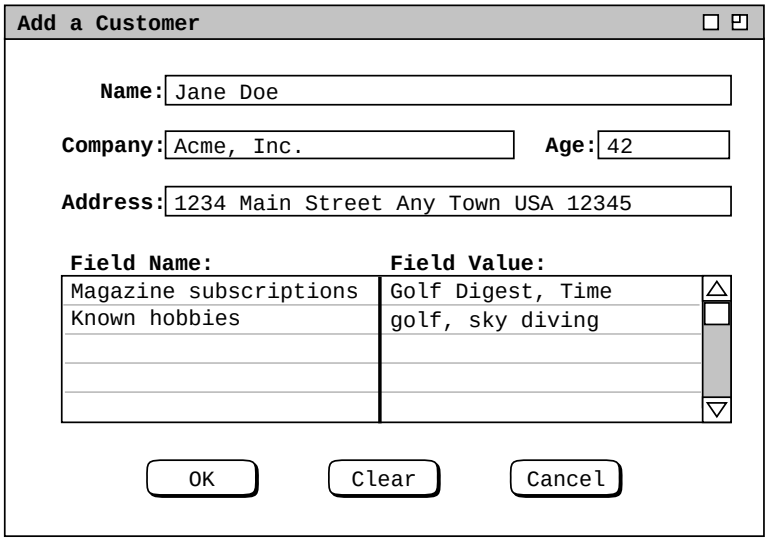


Figure 2: Add customer dialog.

When the user selects the '`Find ...`' item in the `Customers` menu, the system displays the dialog shown in Figure 3. To find a customer, the user can scroll in the name list, or enter the customer name and press the '`Find`' button.

The add-customer and find-customer dialogs are both non-modal. When the user presses `OK` in the add dialog, the find dialog is updated by adding the new customer's name into the displayed list of names.

2.3. Form Letter Generation

When the user selects the '`Create ...`' item from the `Letters` menu, the system displays the edit window shown in Figure 4. The user enters a string value in the '`Template Name`' field, and types the letter template in the '`Template Body`' text area. Figure 4 shows the result of the user having typed in a form letter named "`Golf Club Deal`".

When the user selects '`Generate ...`' from the `Letters` menu, the system displays the dialog shown in Figure 5. The user enters the name of a form letter template from the template data base and fills in desired values for the customer record fields. These values are the letter generation criteria. When the user

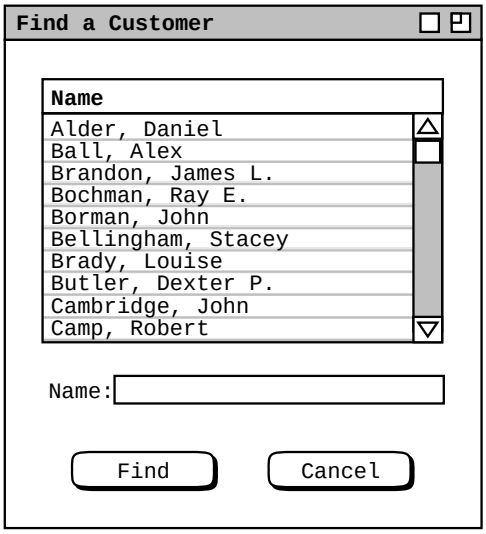


Figure 3: Find customer dialog.

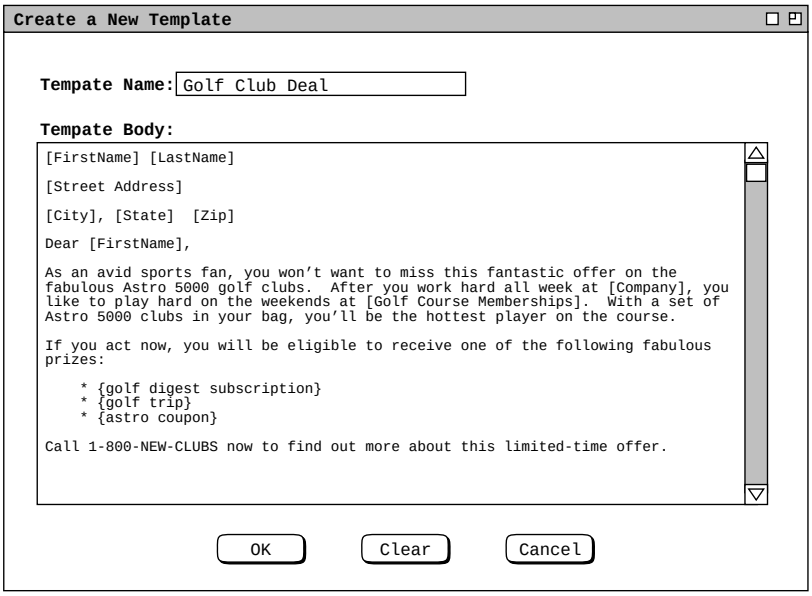


Figure 4: Create template dialog.

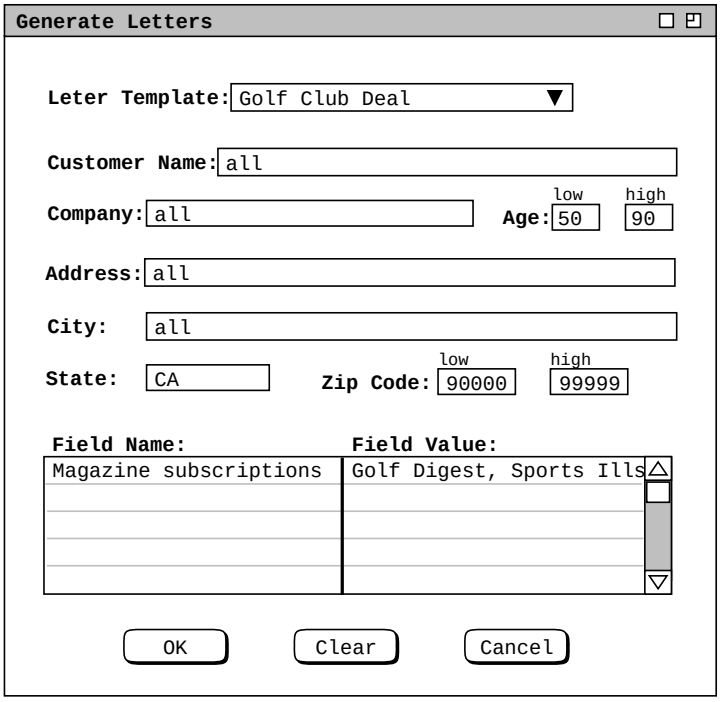


Figure 5: Generate Letters Dialog.

presses the `OK` button, the system generates the letters and prompts the user with a confirmation dialog, indicating how many letters have been generated.

2.4. Options

The items in the '`Options`' menu correspond one-for-one with tabs in an options dialog. When the user selects one of the menu items, the dialog is displayed with the corresponding tab selected. For example, when the user selects the '`Customers`' item in the '`Options`' menu, the system displays the dialog shown in Figure 6.

2.5. File Commands

When the user selects the `Save` item in the `File` menu, the system saves any changes made to any of the three databases. The databases are saved in files with the following fixed names: `customers.dat`, `prizes.dat`, and `letters.dat`.

Further details of file commands are not relevant to the final exam.

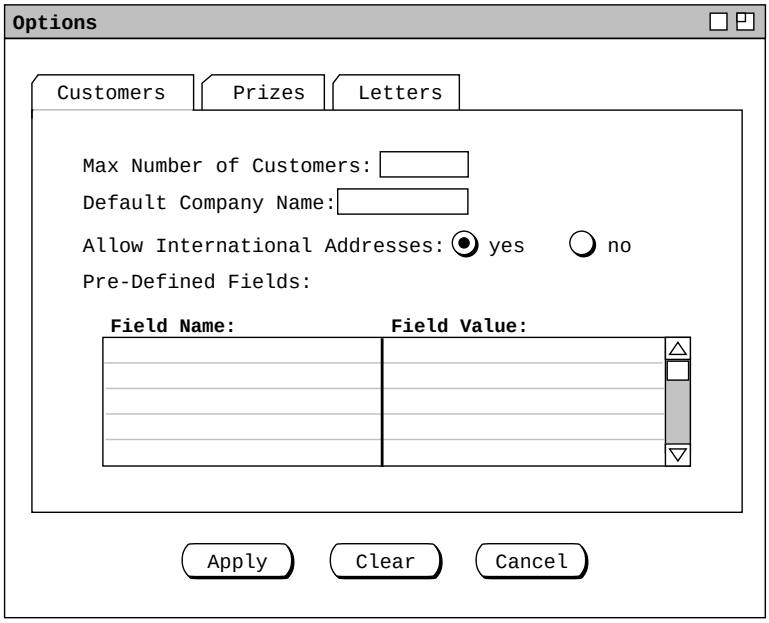


Figure 6: Options dialog.

5. Formal Specification

```
import java.util.Collection;

// In package file:
    /* This package is part of the model, but details of its class definitions
     * are not relevant to the final. */

// In package edit:
    /* This package is part of the model, but details of its class definitions
     * are not relevant to the final. */

// In package letter:
  class LetterTemplateDB {
      Collection<LetterTemplate> data;

      /*@
       requires
          //
         // There is no template in the template db of the same name as the
         // input template.
         //
         ! (\exists LetterTemplate other_template ;
             data.contains(other_template) ;
                 other_template.name.equals(template.name));
       ensures
         //
         // A template is in the output data if and only if it's the new
         // template to be added or it's in the input data.
         //
         (\forall LetterTemplate other_template ;
             \old().data.contains(other_template) <==>
               (other_template.equals(template) ||
                  \old(data).contains(other_template)));
```

```
        @*/
      void createTemplate(LetterTemplate template) { /*...*/ }
      void deleteTemplate(LetterTemplate template) { /*...*/ }
      void editTemplate(LetterTemplate old_template,
          LetterTemplate new_template) { /*...*/ }
      LetterTemplate findTemplate(String templateName)
          { /*...*/ return null; }

        requires
         //
         // A template of the given name is in the template db.
         //
         (\exists LetterTemplate template ;
             data.contains(template);
                 (template.name.equals(criteria.template_name)))

             &&

         //
         // The low and high age values are between 0 and 125.
         //
         ((criteria.low_age >= 0) && (criteria.low_age <= 125) and
          (criteria.high_age >= 0) && (criteria.high_age <= 125))

             &&

         //
         // The low and high zip code values are between 10000 and 99999.
         //
         ((criteria.low_zip >= 10000) && (criteria.low_zip <= 99999) and
          (criteria.high_zip >= 10000) && (criteria.high_zip <= 99999));

        ensures
         //
         // All form letters in the output are based on the template named in
         // the criteria and all of the criteria are me for each letter.
         //
         forall (FormLetter letter ;
             letters.constins(letter) ;
                 matchesTemplate(letter, criteria.template_name)
                    &&
                 criteriaMet(letter, criteria);

      Collection<FormLetter> generateLetters(
          GenerationCriteria criteria) { /*...*/ return null; }

      //
      // Details of the two GenerateLetters auxiliary functions are not
      // necessary for the final.
      //
      boolean matchesTemplate(FormLetter letter , String TemplateName)
          { /* ... */ }
      boolean criteriaMet(FormLetter letten, GenerationCriteria criteria);
        { /* ... */ }
}

  class LetterTemplate {
      String templateName;
      Collection<TemplateBodyItem> body;
  }

  class TemplateBodyItem {
```

```
      boolean isFillIn;
      String value;
  }

  class FormLetter {
      String content;
  }

  class GenerationCriteria {
    String templateName;
    String customerName;
    String company;
    int lowAge;
    int highAge;
    String address;
    String city;
    String state;
    int lowZipCode;
    int highZipCode;
    Collection<CustomField> customFields;
  }

  class CustomField {
      String CustomFieldName;
      String CustomFieldValue;
  }

// In package customers:
  class CustomerDB {
      Collection<CustomerRecord> records;

      void addRecord (CustomerRecord record) {/* ... */}
      void deleteRecord  (CustomerRecord record) {/* ... */}
      void changeRecord(
          CustomerRecord old_record, CustomerRecord new_record) {/* ... */}
      Collection<CustomerRecord> findRecord(String name)
          {/* ... */ return null;}
  }

  class CustomerRecord {
      String name;
      String company;
      int age;
      String address;
      Collection<CustomField> customFields;
  }

// In package prizes:
    /* This package is part of the model, but details of its class definitions
     * are not relevant to the final. */
```

Final Exam Questions:

1. *(16 points)* Draw a class diagram for the model classes derivable from the objects and operations in the `Options` module of the specification (on page 8). Include the following details in the diagram:
   - how and where the MVP `Model` class fits in
   - how and where the top-level Acme tool model class fits in
   - derived class data fields
   - derived class methods, *without signatures*

   NOTE: Show the `Model` class and top-level `Acme` class as one-part boxes, without data field or method details.

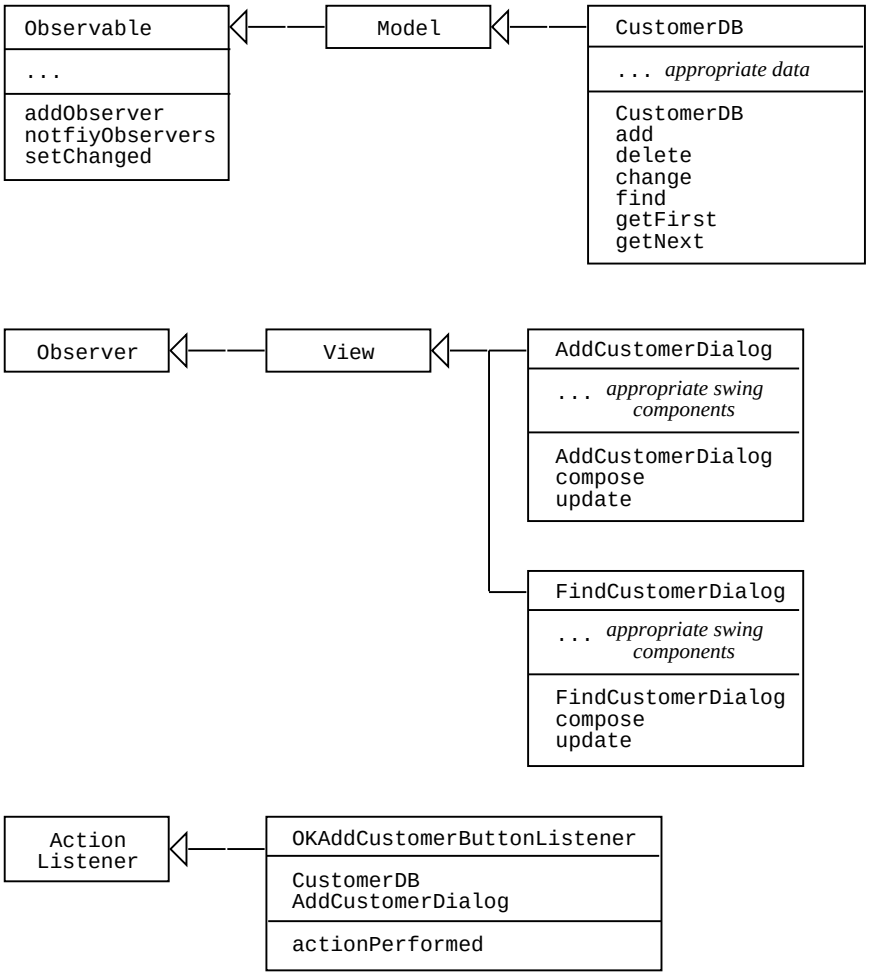
2. *(16 points)* Draw a class diagram for the view classes that define the '`Options`' pulldown menu (Figure 1), the '`Options`' dialog, and the '`Customers`' tab of the options dialog (Figure 6). Include the following details in the diagram:
   - how and where the MVP `View` class fits in
   - derived class data fields, including data fields for the JFC Swing classes used for the components
   - methods to construct and compose the view classes, *without signatures*;

   The following are further clarifications for your answer:
   - Show the `mvp.View` class as a one-part box, without data fields or methods.
   - Define compose methods based on the style used in the Calendar Tool examples, where there are compose submethods for each menu item, and for each row of a dialog.
   - For brevity, you can show multiple components of the same type using the '*' suffix, e.g., "`JTextField*`".
   - You are diagramming the classes that correspond to one menu, one dialog, and one tab of that dialog, not the menubar or any other parts of the GUI.

3. *(16 points)* Consider the last paragraph in Section 2.2 of the requirements on Page 3:

> *The add-customer and find-customer dialogs are both non-modal. When the user presses* `OK` *in the add dialog, the find dialog is updated by adding the new customer's name into the displayed list of names.*

A good way to design and implement this behavior is with the Observer/Observable design pattern. For the purposes of this question, assume the following design has been implemented:



In this design,

a. What method or methods call `CustomerDB.addObserver`?

b. What method or methods call `CustomerDB.notifyObservers`?

c. What method or methods call `CustomerDB.setChanged`?

d. What method or methods call `CustomerDB.getFirst` and `getNext` (these are iterator methods to retrieve all customer records from CustomerDB)?

4. *(20 points)* Give the Java source code for the model class derived from the `LetterTemplateDB` object. Include the following details in the code:
   - the `package` declaration
   - any necessary `import` declarations
   - an appropriate extends clause
   - the class constructor, *including its full signature*
   - derived methods, *including their full signatures*
   - derived data fields

   Leave out the following details from the code:
   - all comments, including pre- and postcondition comments
   - all method body code (i.e., method bodies should be declared as "`{}`")
   - any other class definitions (i.e., give code only for the model class derived from the `LetterTemplateDB` object; you may assume that any referenced classes have been defined.)

   IMPORTANT NOTE: Assume a defensive-style implementation of the `CreateTemplate` and `GenerateLetters` operations, which means that their signatures should include the name(s) of any exception(s) that are thrown. You do not need to define the exception classes themselves, just provide sufficiently mnemonic names for the exceptions.

5. *(20 points)* Give the Java source code for the following two aspects of the `Letters` view implementation:
   a. opening the '`Generate Letters`' dialog from the '`Generate ...`' item of the `Letters` menu
   b. calling the appropriate model method from the `OK` button in the '`Create New Template`' dialog

   To structure your answer, two fill-in code fragments are provided below.

Code fragment for answer to part a:

```
/****
 * Class LettersMenu is the pulldown menu view of the LettersDB model class.  ...
 */
public class LettersMenu extends mvp.View {

    // Constructor and other methods (NO FILL IN NECESSARY)

    /**
     * Add the 'Generate ...' menu item to the menu.
     */
    void addGenerateItem() {

        // FILL IN CODE HERE:









    }

    /** The pulldown menu, initialized in the compose method */
    JMenu lettersMenu;

    /** Reference to Generate Letters dialog, initialized in the constructor */
    GenerateLettersDialog generateDialog;
}

/****
 * Class GenerateLettersDialog defines the dialog for generating a form letter ...
 */
public class GenerateLettersDialog extends mvp.View { /* ... */ }
```

Code fragment for answer to part b:

NOTE: See the next page for additional classes that you can assume exist for your answer.

```
public class OKCreateNewTemplateButtonListener implements ActionListener {

    /**
     * Construct this with the given LetterTemplateDB model and parent dialog
     * view.
     */
    public OKCreateNewTemplateButtonListener(LetterTemplateDB templateDB,
                      CreateNewTemplateDialog dialog) {
       this.templateDB = templateDB;
       this.dialog = dialog;
    }

    /**
     * Respond to a press of the OK button
     */
    public void actionPerformed(ActionEvent e) {

         // FILL IN CODE HERE:









    }

    /** The companion model */
    protected LetterTemplateDB templateDB;

    /** The parent view */
    protected CreateNewTemplateDialog dialog;

}
```

```
/****
 *
 * Class CreateNewTemplateDialog defines the dialog for creating a new form
 * letter template ...
 *
 */
public class CreateNewTemplateDialog extends mvp.View {

    // ...

    /**
     * Return the contents of the template name text field.
     */
    String getTemplateName() { /* ... */ }

    /**
     * Return the contents of the template body text area.
     */
    String getTemplateBody() { /* ... */ }

    /**
     * Display the error messages in the given exception object.
     */
    public void displayErrors(CreateTemplatePrecondViolation errors) { /* ... */ }

}


/****
 *
 * Class CreateTemplatePrecondViolation defines the exception object that is
 * thrown if there is a precondition violation when the createTemplate method
 * is called ...
 *
 */
public class CreateTemplatePrecondViolation extends Exception { /* ... */ }



/****
 *
 * Class LetterTemplate is the model class derived from the LetterTemplate
 * object in the spec.
 *
 */
public class LetterTemplate extends mvp.Model {

    /*
     * Construct this with the given name and body.  The body is supplied as a
     * string which is parsed as necessary to separate text and fill-in fields.
     */
    public LetterTemplate(String name, String body) { /* ... */ }

    public String getName() { /* ... */ }   // Relevant to Question 8

    // ...

}
```

6. *(10 points)* Consider the requirements for the '`File Save`' command in Section 2.5 of the requirements (Page 4). Based on these requirements, fill in the body of the `File.save()` method where indicated below. Assume that the files to which the databases are saved exist in the directory from which the letter system application program was launched. Assume also that CustomerDB, PrizeDB, and LettersDB are serializable.

```
// Assume all necessary imports (NO FILL IN NECESSARY)

/**** Model class for File handling. */
public class File extends Model {

    // Assume constructor and other model methods (NO FILL IN NECESSARY)

    public void save() {
        try {
            // FILL IN HERE:









        }
        catch (FileNotFoundException fnfe) {
            // ... (NO FILL IN NECESSARY)
        }
        catch (IOException ioe) {
            // ... (NO FILL IN NECESSARY)
        }
    }

    // ...

    /** Customer DB */
    CustomerDB customerDB;

    /** Prize DB */
    PrizeDB prizeDB;

    /** Letters DB */
    LettersDB lettersDB;
}
```

7. *(16 points)* Define the companion testing class for the `LetterTemplateDB` model class you derived in your answer to Question 4. Follow the conventions for the testing framework discussed in class. Include the following details:
   - the `package` and `import` declarations
   - the class comment that defines the class testing plan, in terms of the testing phases
   - the class definition, with a completely empty body, i.e., a body defined as just an empty pair of braces -- `{}`

8. *(16 points)* Define the formal specification for the `createTemplate` method that is derived from the `CreateTemplate` operation specified on Page 6. The signature of the method is the one you gave in your answer to Question 4, and any necessary data fields are as defined in your answer to Question 4. Note that you may assume a `getName` method for `LetterTemplate`, as shown on the bottom of Page 15.

9. *(20 points)* Define a unit test plan for the `createTemplate` method. The plan should provide test cases to cover appropriate ranges of input data for the method. (Hint: the important data ranges are the number of text fields and fill-in fields within a template.) Write the plan as it would appear in the tabular-style documentation of the unit testing method; do ***not*** write any of the code for the plan.

10. *(12 points)* Suppose that in addition to the menu-style interface shown in Figure 1 above, a command-line interface was added as an alternate form of user interaction. As a simple example of this style of interface, the find customer command would look like this:

```
> find cust "Doe, Jane"
Name: Jane Doe
Company, Acme, Inc.
Age: 42
1234 Main Street any Town USA 12345
Custom Fields:
  Magazine subscriptions: Golf Digest, Time
  Known Hobbies: golf, sky diving
```

where the first line is the user-entered command, and the lines that follow are system output. Given this added interface, summarize what design changes, if any, would be necessary in each of the following areas:

a. ***package design:***

b. ***model class design:***

c. ***view class design:***

NOTE: The following two questions are general questions about design and version control, not directly related to the previous questions that focused on the Acme spec.

11. *(18 points)* This is a question about SVN conflicts.

    a. Succinctly describe what a SVN conflict is.

    b. How does SVN report a conflict?

    c. In the way we've used SVN this quarter, how should a conflict be resolved? State your answer in terms of a specific file `X.java`, and two project members *P1* and *P2*.