

CSC 307 Final Exam

The exam is open note, open computer. The first five questions are on general topics. The remaining questions refer to a sample requirements specification that is given on Pages 3 through 5 of the exam. Except for Question 9, all of your answers go on the exam pages. There are instructions below for how to submit your answer to Question 9.

The exam lasts up to 170 minutes and there are 170 points possible, so figure roughly 1 minute per point.

1. (3 points) In the process we followed in 307 this quarter, which of the following best describes the *Testing* step (circle the correct answer).
 - a. an ordered step
 - b. a pervasive step
 - c. a managerial step
 - d. a maintenance step
 - e. none of the above

2. (3 points) For a file named "X", what is the SVN command to check if there are any differences between the version of X in your local working directory compared to the version of X in the SVN repository? What I'm asking for is the command that reports whether or not there are differences, **not** the command that reports what the differences are.

3. (4 points) What is (are) the command(s) to properly rename a file named "X" to a file named "Y". The commands can be SVN commands alone, or a combination of SVN commands and UNIX shell commands. What it means to *properly rename* is that the file is renamed in your local directory as well as in the SVN repository.

4. (8 points) In our process of requirements modeling, we begin with developing a set of scenarios that describe user requirements. From these scenarios, we use heuristics to help us derive a model. Briefly describe one such heuristic and give a simple example of its use. The example need not have any pictures, just an explanatory description in words.

5. (20 points) Explain in one to three sentences how it is possible to achieve 100% branch coverage¹ in a test suite, while at the same time having 100% failure of unit tests².

Support your answer by writing an illustrative unit test plan for the following method *m*. When executed, the unit test plan for *X.m* produces 100% branch coverage of the method *m*, with 100% failure of unit testing.

```
public class X {
    public static int m(int i, int j) {
        if ((i > 0) && (i < j)) {
            i++;
        }
        else {
            j++;
        }
        return i+j;
    }
}
```

Give your unit test plan in tabular form, not as code. *Tabular form* is described in the 307 lecture notes as a four-column table that has a list of test cases, with column headings for (1) the test case number, (2) the case input, (3) the expected output, (4) explanatory remark for the test case.

¹ *Branch coverage* is defined as fully exercising the boolean logic of all program branching statements. This form of coverage is measured by Cobertura, and similar tools.

² Assume that a failed unit test does not abort the entire test suite. That is, if a unit test fails, the remaining unit tests that follow it will still execute.

Given below are excerpts from the requirements specification for a simple form letter mailing system. Following the requirements specification excerpt are the remaining final exam questions. These questions all refer to the requirements spec.

1. Introduction

In an effort to survive a few more years in the paper junk mail business, Acme Junkmail Inc. needs to make their business more efficient. Acme specializes in producing individualized advertising letters. The letters are mailed to potential customers who may be interested in purchasing certain products. At present, Acme has fifty employees who produce the letters using standard office software. 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 an appropriate identifier.

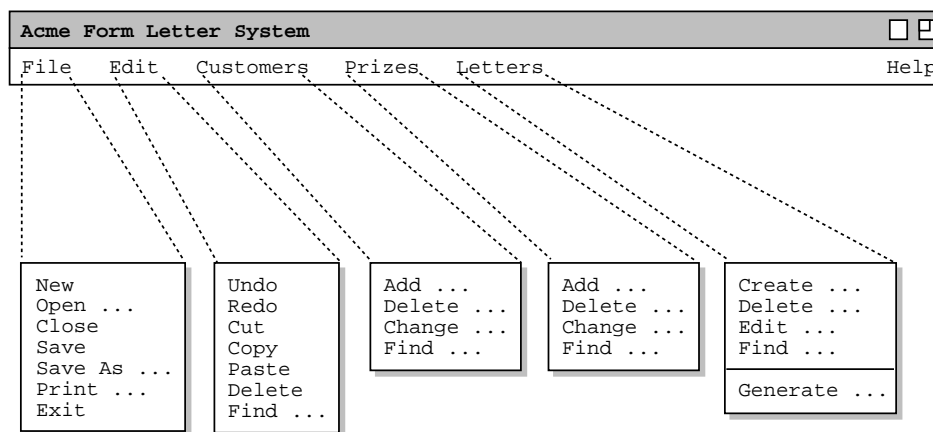


Figure 1: Expanded command menus.

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.

2.2. Customer Database Management (Abbreviated Requirements)

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.

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.

Field Name:	Field Value:
Magazine subscriptions	Golf Digest, Time
Known hobbies	golf, sky diving

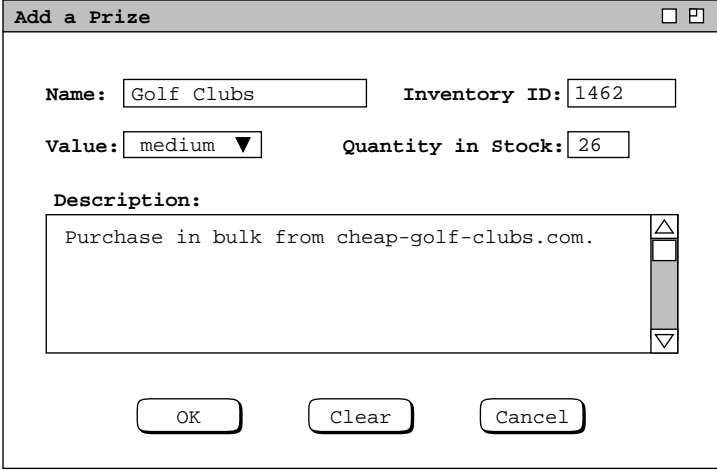
Figure 2: Add customer dialog.

Figure 3: Find customer dialog.

2.3. Prize Database Management (Abbreviated Requirements)

When the user selects the 'Add . . .' item from the Prizes menu, the system displays the dialog shown in Figure 4. The Name field is a free-form string; it cannot be empty. The Inventory ID is a positive integer; it also cannot be empty. The Value field is one of 'low', 'medium', or 'high'. The Quantity in Stock is a positive integer; it can be empty, which indicates a quantity in stock of 0. The Description field is an optional free-form string.

When the user presses the 'OK' button in the add prize dialog, the system checks that no prize of the given Inventory ID is already in the prize database. If there is no such prize, then the system adds a prize with the given information. Otherwise, the system displays an error dialog informing the user that the specified Inventory ID is already in use.



The screenshot shows a dialog box titled "Add a Prize". It contains the following fields and controls:

- Name:** A text input field containing "Golf Clubs".
- Inventory ID:** A text input field containing "1462".
- Value:** A dropdown menu with "medium" selected.
- Quantity in Stock:** A text input field containing "26".
- Description:** A text area containing "Purchase in bulk from cheap-golf-clubs.com." with a vertical scrollbar on the right.
- Buttons:** Three buttons at the bottom: "OK", "Clear", and "Cancel".

Figure 4: Add prize dialog

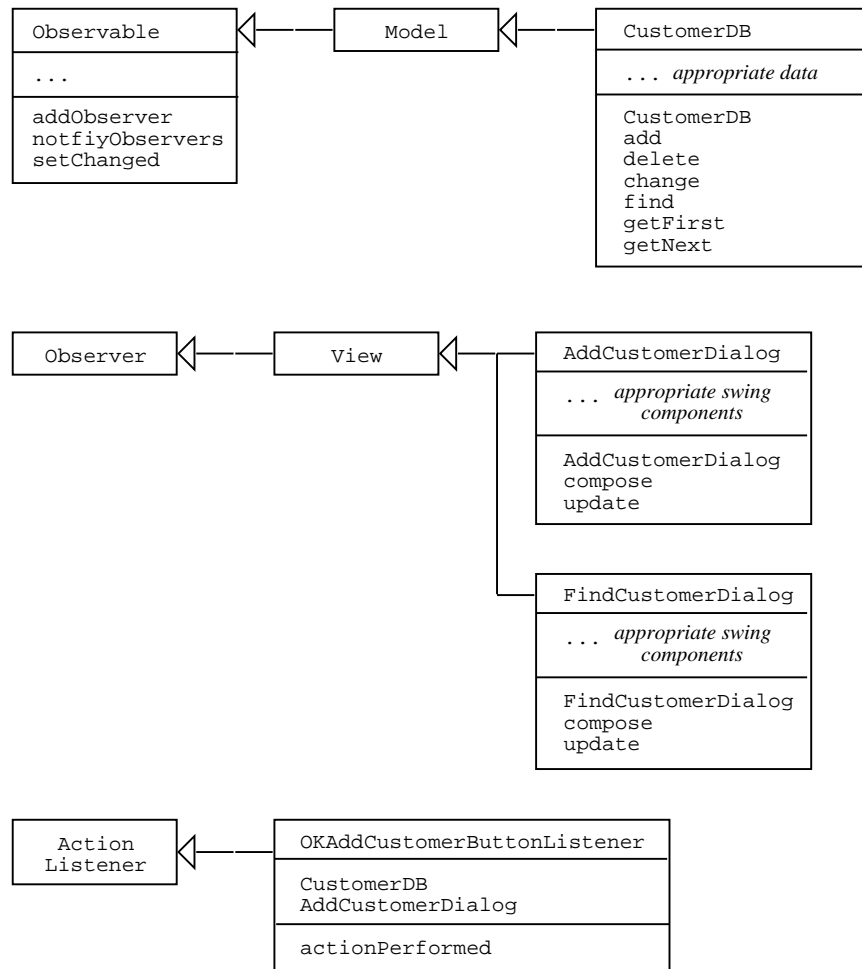
6. (10 points) Draw a package diagram for the Acme Form Letter System, based on the preceding requirements excerpts. Follow the Information Process Tool (IPT) packaging pattern we've discussed in class, or the modified version of the IPT pattern you used for your team's project this quarter. In either case, the package diagram should reflect a separation of the model and view components of the design.

7. (6 points) In the requirements excerpts on Pages 4 and 5, the section headings include the note "(Abbreviated Requirements)". This indicates that some UI pictures and narrative have been omitted. Based on the style of requirements we wrote this quarter, what would you say is the most significant omission from these abbreviated requirements?

8. (20 points) Consider the last paragraph in Section 2.2 of the requirements on Page 4:

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,

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

9. (80 points) For this question of the exam, you are to implement model classes for the Acme Prize DB described on Page 3 of the exam, and view classes for Add Prize Dialog described on Page 5. In particular, you must provide a concrete implementation of the following abstract `PrizeDB` class, along with the `Prize` class which it uses.

```
import java.util.Collection;

abstract class PrizeDB {

    /** The prizes in this DB. */
    Collection data;

    /** Add the given Prize p to this.data. */
    public abstract void add(Prize p);

    /** Delete the prize of the given ID from this.data */
    public abstract void delete(int id);

    /** Change the prize of the given id to the given Prize p */
    public abstract void change(int id, Prize p);

    /** Return the prize of the given id, null if no such prize */
    public abstract Prize find(int id);

}
```

The following are precise details of model and view implementation:

- the abstract `Collection` interface must be replaced by some concrete collection class; it can be as simple as `ArrayList`, or the equivalent
- the `PrizeDB.add` method must add the prize to the the collection
- to provide a quick form of validation, the `add` method must also print the size of the collection to `stdout`
- the `PrizeDB.add` method must have a Spest comment the defines the input constraints specified in paragraphs 1 and 2 of Section 2.3 on Page 5 of the exam; namely:
 - the name field of the prize is not empty
 - the inventory ID and quantity fields of the prize are positive integer values, i.e., strictly greater than 0
 - there cannot be a prize in the input database with the same ID as the prize to be added
- The Spest comment must also have a postcondition that specifies that the prize is properly added to the prize DB, where "properly" means that the prize is added, no existing prizes are removed or modified, and no extra prizes are added; i.e., the "no junk, no confusion" logic we discussed in class
- when the OK button is pressed in the add prize dialog, the values entered in the dialog fields are extracted from the dialog, a new prize is created, and `PrizeDB.add` is called to add the prize to the database
- after OK is pressed, the dialog stays in place on the screen, so multiple prizes can be added
- the implementation must have model/view separation, i.e., the model and view code must be in separate classes

The following are specific simplifying assumptions about the model implementation:

- neither the `PrizeDB.add` method nor the `Prize` constructor need implement any data validation constraints; if the user enters invalid data into one or more fields of the add prize dialog, the program is allowed to have undefined behavior; in other words, the program does not need to provide code that validates the specification defined in the Spest comment
- the implementations of `delete`, `change`, and `find` `PrizeDB` methods can be completely empty, except for the `return null` required to allow `find` to compile properly
- the implementation does not have to be organized into separate packages; that is, all classes may be in the same package directory, and there need be no explicit `package` declarations in any of the classes

- no class or method comments are necessary

For the view, you must provide an implementation that displays the dialog in Figure 4 on Page 5 of the exam. As you see fit, you may make the following simplifying assumptions to ease the implementation of the dialog:

- the Value field can be a string instead of a drop-down
- the Description field need not have a scroll bar
- the Clear and Cancel buttons may be eliminated
- the spacing and alignment of the dialog fields need not be exactly as shown in Figure 4 on Page 5, but the overall horizontal/vertical layout must be preserved

Your implementation must provide some form of main program driver that constructs a Prize DB and displays the Add Prize dialog on the screen. You may use the GUI toolkit and IDE that you employed this quarter to define the driver program. The bottom line is that the program must be executable, display the dialog when run, and the dialog must communicate with the Prize DB model as described above.

You must use the `handin` program on `unix3` to submit your answer for this question. Submit to the user `gfisher` and the submission directory `307_final`. Hence, your `handin` command looks like this

```
handin gfisher 307_final files ...
```

You must submit all source files for your program plus a README file that describes *precisely* how to run your submission. The following is a concrete example of submitting a program that uses Java Swing for the implementation:

```
handin gfisher 307_final Main.java PrizeDB.java Prize.java AddPrizeDialog.java README
```

with the README file saying "Compile using `javac *.java` and run with `java Main`." If it is convenient for you to produce an executable jar file, you may do so. In this case, the submission could look like

```
handin gfisher 307_final Main.java PrizeDB.java Prize.java AddPrizeDialog.java Main.jar README
```

with the README saying "Run with `java -jar Main.jar`." Other variants of submission are acceptable, as long as you submit all of your source files and make it absolutely clear how to run your program.

10. (16 points) Define the companion testing class for the `PrizeDB` model class you implemented in your answer to Question 9. Include the following details:
- the class header, with its name and any class it extends
 - the class comment that defines the class testing plan, in terms of the testing phases
 - the names and signatures of unit test methods, but otherwise empty; i.e., the unit test methods need not have comments and their bodies are empty

Note that you *are not* required to submit this test file with `handin`; just put your answer here on paper, using the back of the page as necessary.