

CSC 307 Midterm Exam

*Instructions: The exam is open-note. It is worth a total of 125 points.
Use back of pages as necessary for your answers.*

1. Software Process Questions

- a. (4 points) Describe a software project "stakeholder" in 25 words or less.

Anyone who has some interest, large or small, in a software product.

Full credit for some variant of this.

Half credit for noting some specific interest only, e.g., financial.

- b. (5 points) Give two key differences between the traditional process we are following this quarter and an Agile software process (e.g., Extreme Programming).

A complete requirements document is not developed in an agile process.

There is more frequent process iteration in an agile process

Other acceptable answers include any derivable from Notes 1-2 or Section 2.6.5 of the text.

All or nothing for some variant of this.

- c. (4 points) Why is Testing called a "pervasive step" in the software process we are following?

Because it is conducted at regularly-scheduled intervals during all phases of development.

Full credit for some variant of this;

Half credit for something reasonable, but not fully correct.

- d. (4 points) What is a significant problem with not having Testing be a pervasive step?

If Testing is left until the end of the entire development process, following implementation, errors may well be harder to find. For example, it may be quite hard to find errors in untested requirements when looking at Java code that attempts to implement the requirements.

- e. (3 points) Formal specifications are formal enough to be mechanically analyzed for completeness and consistency (true or false)?

true

2. Requirements Analysis Questions

Speaking as your customer, I want to add a user complaint feature to the tool that your team is writing the requirements for this quarter. I want the following features:

- users can send a complaint directly from the tool, without having to launch an external email program;
- if users want a response, they can optionally include their email address along with their name and the text of the complaint; note that it's only the email address that is optional, i.e., the complaint text and name are required, with or without email address;
- when a complaint is sent, the user receives confirmation of the date and time of sending, plus a polite message that the complaint will be looked into by a responsible party; the time of the complaint is reported in hours, minutes, and seconds
- the tool maintains a list of submitted complaints; users can view the complaints that they and other people have sent, including complaint responses if they exist.
- the user can remove a previously sent complaint by giving the date and time the complaint was sent

Given these requirements, a responded-to complaint consists of the following elements: the name of the sender, an optional email address, the date/time of sending, the text of the complaint itself. When and if a complaint is responded to, the response text is added to the complaint.

I'd like the complaint sending interface to be as simple as possible. In particular, I don't want the user to have to enter an address for where the complaint is to be sent, or any other administrative information, that the system can determine on the user's behalf. The tool knows how and where to send the complaint to a responsible party who can deal with it and respond if necessary.

- a. (16 points) Describe where the complaint-related commands best fit in your tool's UI. Your answer entails determining what the top-level complaint commands are, based on the requirements outlined above.

Support your description with a succinct UI picture, showing the top-level access to the complaint commands in your tool. By "top-level access", I mean show where in your tool's main GUI the complaint commands appear. By "succinct" I mean show as little non-essential detail as possible in the picture.

If your tool has more than one end-user interface, describe how the commands are added to what you consider to be the primary interface, or the interface where the commands make the most sense.

I'd put the commands in the Help menu of the regular-user Calendar Tool UI. Here's a picture of the Help menu:

```

Help
-----
About
Quick Help
Detailed Help ...
-----
Send Complaint ...
View Complaints ...

```

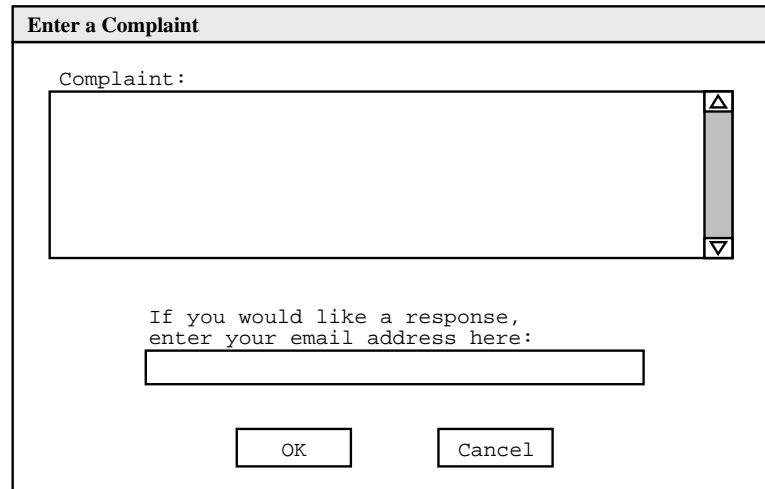
Grading Notes:

- 6 pts for each command
- 4 pts for decent description and reasonable placement
- Placement on the file menu would be fine too, as would placement in some other reasonable part of the top-level GUI, though the menubar is favored per the class discussions about it being the most expected place to find major commands.
- -3 pts for putting it in more than one place

- b. (36 points) Write a requirements scenario describing the command that sends a complaint. Write it precisely in the style presented in the lecture notes and examples, including being precise about the narrative prose style. Include as many screenshots as you think are necessary. Describe fully all user interactions, including input value constraints. **Hint:** More than three or four screens is likely too many.

Note well: This scenario has nothing to do with the complaint viewing, complaint removing, or with complaint processing after it is sent. It just covers complaint sending.

When the user selects the 'Send Complaint' command in the 'Help' menu, the system responds by showing the dialog in Figure 1.



Enter a Complaint

Complaint:

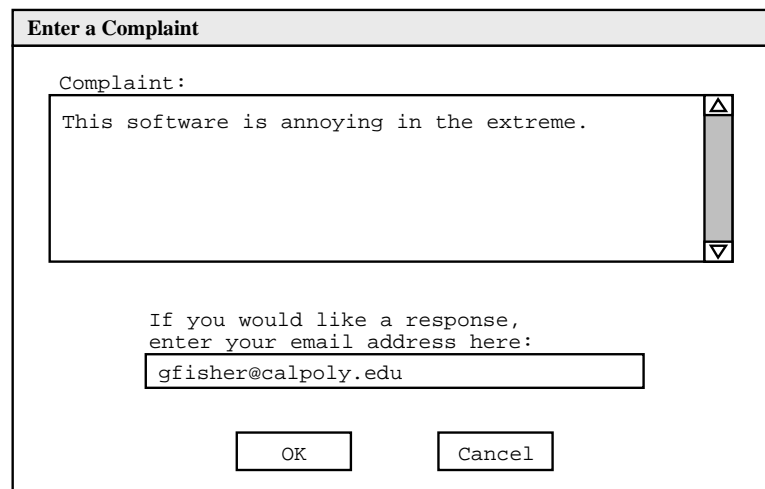
If you would like a response,
enter your email address here:

OK Cancel

Figure 1: Complaint entry dialog.

The dialog contains a place for the user to enter a complaint, and an optional email address. The complaint is a free-form text string. The email address is not validated in any way. This scenario assumes that the user has already been identified by the Calendar Tool, so the user name associated with the complaint is available from the Calendar Tool workspace. The scenario also assumes that the Calendar tool uses the system date and time for the complaint.

Figure 2 shows the result of the user having entered a complaint text and email.



Enter a Complaint

Complaint:

This software is annoying in the extreme.

If you would like a response,
enter your email address here:

gfisher@calpoly.edu

OK Cancel

Figure 2: Complaint entry dialog filled in.

Figure 2 shows a typical user complaint. When the user presses OK, the system sends the complaint to the appropriate processing personnel. The system then responds with the confirmation dialog shown in Figure 3.

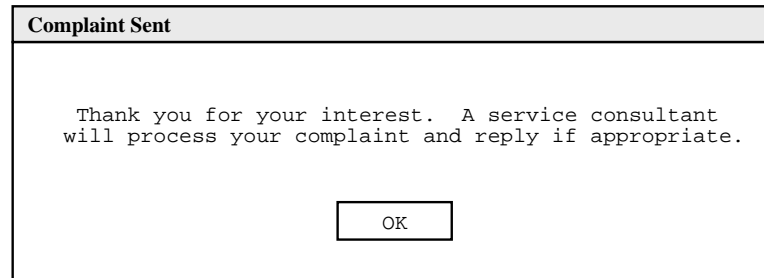


Figure 3: Complaint sent confirmation.

Grading Notes:

- 15 pts for initial dialog picture (8 pts) and explanation of contents (7 pts), with these specific deductions:
 - o -2 for no window title
 - o -2 each for missing name, complaint text, and/or email
 - o -2 each for missing confirm and/or cancel buttons
- 14 pts for filled-in dialog, with 7/7 split and comparable deductions
- 7 pts for confirmation, including "polite" message, with these deductions:
 - o -2 pts for description of result that message is sent to a "responsible authority" of some kind
 - o -5 for missing confirmation is some screen form, most typically a dialog
- -3 pts for each prose style violation, up to -9
- -3 pts for each egregious misplacement of figures, up to -6

3. Object Modeling Questions

- a. (20 points) Based on the complaint features outlined on Page 2 and your answers to Questions 2a and 2b, define an abstract Java model to represent the following aspects of complaint management: the complaint object; the cumulative list of complaints sent by all users; the operations to send, remove, and view complaints. Include full signatures for the operations. No comments are necessary.

```
import java.util.List;

abstract class Complaints {
    List<Complaint> list;
    abstract void send(Complaint complaint);
    abstract void remove(DateAndTime dateAndTime);
    abstract List<Complaint> view();
}

abstract class Complaint {
    String name;
    String email;
    DateAndTime dateAndTime;    // using java.util.Date is acceptable
    String text;
    String response;           // acceptable as data field here or in subclass,
                               // but not both places
}

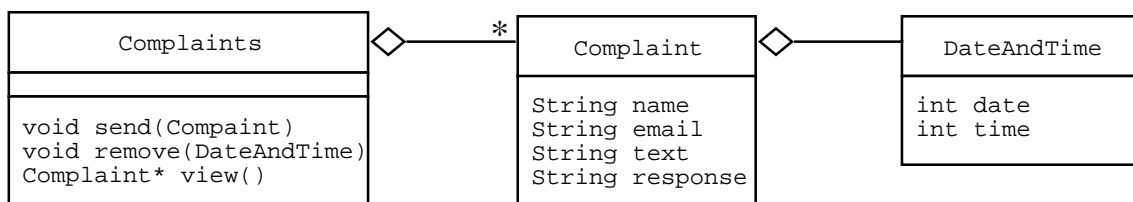
// Also acceptable is to have response field in a subclass, like this:
abstract class ComplaintWithResponse extends Complaint {
    String response;           // acceptable as data field here or in Complaint class,
                               // but not both places
}

// Other variants are acceptable, including use of java.util.Date with no
// separate class defined here
abstract class DateAndTime {
    int date;
    int time;
    // Per the note on the test, we assume this compareTo works properly
    abstract int compareTo(DateAndTime other);
}
```

Grading Notes:

- 1 pt for import
- 8 pts for Complaints class (2 for header, 3 for collection, 3 for method signature)
- 8 pts Complaint class (1 for header, 1 each for data fields)
- -2 pts for each significant inconsistency with the scenarios, up to -6; notable inconsistencies are missing components and substantial deviations in object names compared to the scenarios

- b. (8 points) Draw a UML diagram that is equivalent to the preceding Java definition.



Grading Notes:

- or other equivalent form
- method signature and field names are optional

c. (25 points) Using Sppest notation, define preconditions and/or postconditions that formally specify the following requirements for the Java model you defined in question 3a:

- a complaint sender's name must be greater than or equal to one character in length, and less than or equal to 100 characters in length
- if a sender's email address is not empty, then the email address must contain exactly one '@' character; (partial credit for *at least one* '@' character)
- for the complaint viewing operation, the list of complaints is sorted by date and time sent, from earliest to latest; you may assume the existence of a properly implemented `compareTo` method for whatever representation of date and time you used in your Java model

It's up to you to determine which method or methods these conditions are specified for, and whether they are preconditions, postconditions, or both. Once you have made these determinations, write your answer by giving the full signature of the method(s). Above each signature, write the appropriate Sppest specification.

```

/*
  pre:
    complaint != null &&
    complaint.name != null &&
    complaint.name.length() >=1 && complaint.name.length() <= 100 &&
    if (complaint.email != null && !complaint.equals(""))
      exists (int i; i>=0 && i<complaint.email.length();
        complaint.email.charAt(i) == '@' &&
        forall (int j; j!=i && j>=0 && j<complaint.email.length();
          complaint.email.charAt(j) != '@');
*/
abstract void send(Complaint complaint);

/*
  post:
    forall (int i; i>=0 && i<\result.size()-1;
      return.get(i).dateAndTime.compareTo(
        return.get(i+1).dateAndTime) < 0);
*/
abstract List<Complaint> view();

```

Grading Notes:

- 1 pt for null check of complaint
- 1 pt each for null checks of complaint.name and complaint.email
- 5 pts for first condition, 9 pts for second condition, 8 pts for third condition
- -4 pts for "at least one" instead of "exactly one"

Additional Note: The next page has the fully compilable model, including preconditions and postconditions, plus some additional comments. What's on the next page is not an additional part of the expected student answer; it's just there to provide further explanatory information.

```

import java.util.List;

abstract class Complaints {
    List<Complaint> list;
    /*
    pre:
        // Complaint itself can't be null
        complaint != null &&

        // Sender's name length must be >= 1 and <= 100
        complaint.name != null &&
        complaint.name.length() >=1 && complaint.name.length() <= 100 &&

        // If the email address isn't empty then
        if (complaint.email != null && !complaint.equals(""))

            // the address contains at least one '@'
            exists (int i; i>=0 && i<complaint.email.length();
                complaint.email.charAt(i) == '@' &&

                // and exactly one '@' (i.e., with '@' at position i,
                // for all j!=i, there's no '@' at position j)
                forall (int j; j!=i && j>=0 && j<complaint.email.length();
                    complaint.email.charAt(j) != '@'));

    */
    abstract void send(Complaint complaint);

    /* No Spent for this method */
    abstract void remove(DateAndTime dateAndTime);

    /*
    post:
        // This is sorting the logic from the lecture notes, suitably
        // adapted to work here.
        forall (int i; i>=0 && i<return.size()-1;
            return.get(i).dateAndTime.compareTo(
                return.get(i+1).dateAndTime) < 0);

    */
    abstract List<Complaint> view();
}

abstract class Complaint {
    String name;
    String email;
    DateAndTime dateAndTime;    // using java.util.Date is acceptable
    String text;
    String response;           // acceptable as data field here or in subclass,
                               // but not both places
}

// Also acceptable is to have response field in a subclass, like this:
abstract class ComplaintWithResponse extends Complaint {
    String response;           // acceptable as data field here or in Complaint class,
                               // but not both places
}

// Other variants are acceptable, including use of java.util.Date with no
// separate class defined here
abstract class DateAndTime {
    int date;
    int time;
    // Per the note on the test, we assume this compareTo works properly
    abstract int compareTo(DateAndTime other);
}

```

