CSC 308 Lecture Notes Weeks 9 and 10
Modeling Idioms
Requirements for File and Edit Commands
Non-Functional Requirements

I. Milestone 8 -- *REVISED*.

A. Due 11:59PM Wednesday 11 March

B. The deliverables are:

1. Prototype, three prototype GUIs per team member
2. Model updates, including Spest checking
3. ***NO*** Requirements updates for Milestone 8, to be completed for Milestone 10

II. Milestones 9 and 10 -- see the handout.

A. Requirements, model, and prototype updates.

B. Add section on data storage and copy/paste requirements.

C. Add section on error conditions.

D. Add section on Non-Functional Requirements, specifically Sections 3.1 and 3.2 as described in the milestone 10 writeup.

E. If necessary, add Appendix A: Late Updates.

F. If necessary, and `administration/project-summary.html`

III. Modeling idioms.

A. The next several items in the notes discuss common modeling forms, of use in one or more 308 projects.

B. These idioms can be used as appropriate to a particular 308 project, as the JML is refined.

IV. Authenticated access to stored data.

A. General idea:

1. Users with IDs and passwords on a particular data server are allowed access to the data, if they authenticate their identity properly.
2. A login operation performs the authentication, and if successful delivers the shared data, otherwise not.
3. This login is defined as the only access to the data, i.e., the only operation that delivers the data from that server as an output.

B. Usage in 308 could includes Grader access to SIS data, TestTool access to shared question bank, guest access to an EClass lecture or CSTutor tutorial.

C. A basic example:

```
import java.util.Collection;



/**
 * A server has a list of authentic users and some data.
 */
abstract class Server {
   Collection<UserRecord> users;
   Data data;
```

```
    /**
     * Login to this server to access its data.
     *
      pre:
        exists (UserRecord user ;
            users.contains(user) ;
                user.id.equals(id) &&
                    user.password.equals(password));
      post:
        return.equals(data);
     */
    abstract Data login(String id, String password);
}



/**
 * A UserRecord is an ID/Password pair.  A more elaborate
 */
abstract class UserRecord {
    String id;
    String password;
}
```

```
/**
 * Data is a purely abstract class that represents the
 * data stored on a server that’s accessible via login.
 */
abstract class Data {}
```

V. **Serial navigation through ordered collections.**

A. General idea:

1. A sequence of items is modeled as a list, and a current position in the list.
2. Next/previous navigation is modeled as increment/decrement of the current position value.

B. Usage in 308 includes CSTutor page navigation, EClass slide navigation and any place where the user can traverse with Next/Prev commands through a list of items.

C. A simplified example:

```
import java.util.List;



/**
 * A sequence of items index from 1 to items.size().  The
 */
abstract class Sequence {
    List<Item> items;
    int cur_item;
```

```
    /**
     * Move to the next item in the sequence if not at the
     * last item.

      pre: cur_item < items.size();
      post: cur_item’ == cur_item + 1;
```

```
         */
        abstract void next();




        /**
         * Move to the previous item in the sequence if not at
         * the first item.

          pre: cur_item > 1;
          post: cur_item’ == cur_item - 1;

         */
        abstract void prev();

    }



    /**
     * An item in a sequence.
     */
    abstract class Item {}
```

VI. **HTML Content**

A. General idea:

1. HTML markup is modeled as tags and text interspersed.
2. Only as many tags as are needed by a particular tool are modeled.
3. The full specification of HTML is referred to at a definitive source, in this case `w3.org`.

B. Usage of this kind of object model in 308 includes the HTML content in an EClass presentation, a TestTool test question, a CSTutor page.

C. Here’s a simplified model of HTML content:

```
import java.util.List;



/**
 * HTML content consists of command tags and text.
 */
abstract class HtmlContent {
    List<TagAndValue> content;





    /**
     * Render this into the format displayed to a user.
     * See the <a href=
     * "http://www.w3.org/TR/html5/semantics.html"> W3C
     * </a> the for the full specification of how this
     * rendering is performed.
     */
    abstract RenderedHTMLContent display();

}
```

```
/**
 * Class TagAndValue has an enumerated tag value and
 * string value.
 */
class TagAndValue {
    Tag tag;
    String value;
}
```

```
/**
 * The Tag enumeration consists of the tags that need to
 * be modeled for a particular tool’s use of HTML.
 */
enum Tag {
    TEXT, TAG1, TAG2, /* ... */
}
```

```
/**
 * As defined by W3C at <a href=
 * "http://www.w3.org/TR/html5/semantics.html".>
 */
abstract class RenderedHTMLContent {}
```

VII. **Recursive object definitions.**

A. General idea:

1. We have a nested structure that goes to an indefinite depth.
2. This is well modeled with a recursive definition.

B. Usage in 308 includes the graded item hierarchy in the Grader project, the lecture topic hierarchy in EClass.

C. Here is a simplified example of a spreadsheet, with indefinitely nested columns:

```
import java.util.Collection;
```

```
/**
 * A generic spreadsheet consists of rows.
 */
abstract class SpreadSheet {
    Collection<SpreadSheetRow> rows;
}
```

```
/**
 * A row has a number and a collection of columns.
 */
abstract class SpreadSheetRow {
    int number;
    Collection <Column> columns;
}
```

```
/**
 * A column has a heading, some data, and possibly
 * subcolumns.  When subcolumns are non-nil, the data is
 * typically some combination of the subcolumn data
 * values, such as the sum if data values are numeric.
 */
abstract class Column {
    String heading;
    Data data;
    Collection<Column> subColumns;
}



/**
 * Whatever kind of data can appear in spreadsheet cells.
 */
abstract class Data { /* ... */ }
```

VIII. **Wizards, and other strictly-ordered operation sequences.**

A. General idea:

1. The operational model for a wizard-style interface is to require sequential execution of the wizard steps.
2. This can be modeled by having each step of the wizard be an operation that produces a unique output, that is the required input to the next step.

B. Usage in 308 is anywhere that a wizard-style UI is presented to the user.

C. Here’s a generic example:

```
/**
 * Class Wizard has operations that must be performed in
 * sequential steps.  To enforce this sequential behavior,
 * each wizard step takes a unique type of input and
 * produces a unique type of output.
 *









 *
 * When the types of the i/o objects are unique, then the
 * requirement that wizard step N must follow step N-1 is
 * enforced by operation stepN being the only operation
 * that accepts an input of type StepMinus1Ouput
 *









 *
 * If output types themselves are not unique type, a
 * postcondition on stepNMinus1 can set a component to a
 * unique value that a precondition on stepN checks, to
 * ensure the data in fact come from step 1.
 *
 */
abstract class Wizard {
    /**
     * Perform wizard step N, producing output to be given
     * to wizard step 2.
     */
    abstract Step1Output step1(Step1Input in);







    /**
     * Like step1.
     */
    abstract Step2Output step2(Step1Output in);







    /**
     * Like steps 1 through N-1.
     */
    abstract StepNOutput stepN(StepNminus1Output in);
}






abstract class Step1Input { /* ... */ }
abstract class Step2Input { /* ... */ }
abstract class StepNInput { /* ... */ }

abstract class Step1Output { /* ... */ }
abstract class Step2Output { /* ... */ }
abstract class StepNminus1Output { /* ... */ }
abstract class StepNOutput { /* ... */ }
```

IX. Summing.

A. General idea:

1. It is sometimes necessary in an operational model to specify a specific arithmetic computation, such as a sum of values from a list.

2. In the functional modeling language we are using, such computations are defined using recursive auxiliary functions.

B. Usage of a summing function is necessary in the 308 Grader tool, in specifying the operation that displays the pie chart or histogram of grades. Summing may also be necessary in the TestTool for computing a sum of student scores on tests.

C. Here is the definition of an auxiliary function `SumList`, that can be used in a postcondition that specifies that the output of an operation is based on the sum of elements in a list.

```
import java.util.List;

/**
 * The recursive definition of the SumList function uses the following
 * strategy:
 *
 *     (a) If the list contains no elements, then the sum is 0
 *     (b) Otherwise, recursively compute the sum of the first list element
 *         with the sum of the rest of the elements
 *
 * The "rest of" a list is denoted by the expression
 *
 *     l[2:#l]
 *
 * The colon-separated expression between the square brackets denotes a
 * sublist of the form
 *
 *     start:end
 *
 * where ’start’ and ’end’ are the index positions of the sublist.  The ’#’
 * operator is list length.  Hence, the sublist [2:#l] extends from the second
 * position of the list to the last position.  I.e., it’s the "rest of" a list
 * beyond position 1.
 *
 */

class IntList {
    List<Integer> data;

    int sum(List<Integer> data) {
        if (data.size() == 0) return 0;
        else return data.get(0) +
            sum(data.subList(2, data.size() - 1));
    }
}
```

X. **Undo/redo.**

A. General idea:

1. Almost all user-oriented software these days has undo/redo commands.
2. A simple abstract model of undo/redo can be defined by saving a full copy of any changed data whenever a data-changing operation is performed.
3. The undo operation then restores the saved data.
4. This kind of model is almost always too inefficient to implement directly, since it involves copying potentially large amounts of data.
5. The model does however precisely define the meaning of undo/redo, and as such is another good example of where a model need not address algorithmic efficiency issues, as long as it accurately defines operational behavior.

B. Here is an example of a simple one-level undo/redo model, defined in terms of a generic tool workspace and a generic operation that changes tool data.

```
/**
 * Simplified data model for a tool workspace.
 */
abstract class ToolWorkspace {
    ToolData data;
    ToolData undo_data;
    ToolData redo_data;
```

```
    /**
     * To perform some tool operation, set the data of the
     * output workspace to the new data, and the undo data
     * of the output to the original data in the input
     * workspace.

       post:
         undo_data’.equals(data) &&
         data’.equals(new_data) &&
         redo_data’ == null;
      */
     abstract void some_operation(ToolData new_data);



    /**
     * If some tool operation has been performed that sets
     * undo data in the workspace, then the effect of undo
     * is to set workspace data to that undo data,
     * otherwise undo has no effect.

       pre: undo_data != null;
       post: data’.equals(undo_data) &&
             undo_data’ == null &&
             redo_data’.equals(data);
      */
     abstract void undo();
 }




 /**
  * Whatever data for which operations are undoable.
  */
 abstract class ToolData { /* ... */ }
```

XI. Specifying a "good" computed result, or the "best" result.

A. At times in user requirements, an output needs to be measured as good or the best of any possible result.

B. In terms of the operational model, such a result is most typically stated as a postcondition.

C. The postcondition can be stated in two major parts:

1. The output meets a basic set of requirements.
2. The output is also minimal or maximal in some sense.

D. In 308, such postconditions can be specified for the generated schedule in the Scheduler tool, or a generated test in the TestTool.

E. A detailed of example "best result" specification from the Calendar Tool, appears later in the notes.

XII. **Requirements for file and edit commands.**

A. Following the requirements scenarios for the major commands of your system, cover any remaining details of file and edit commands.

B. Put these details in one or two sections following the other functional requirements sections.

C. For file commands, consider clearly what objects are saved to and opened from disk.

D. For edit commands, consider clearly what objects are operated on by the cut, copy, and paste commands.

XIII. **Requirements for error conditions**

A. Following the file and edit command section, include a section that covers error conditions.

B. You need only summarize the error conditions you’ve covered already in earlier scenarios.

C. For error conditions that have not yet been covered, show and describe the error display screens.

D. You need only show one version of each generic error screen, listing the different error message texts.

XIV. **Specification of error conditions.**

A. In the 308 specification methodology we do this with preconditions.

B. From the user’s perspective, each clause of a precondition corresponds to an error if the clause is not true.

C. The error conditions described in the requirements correspond directly to violation of precondition clauses.

D. We’ll discuss the formalization of this further next week.

XV. **Other requirements.**

A. Help -- not required for 308.

B. Other GUI details -- not required for 308.

C. Data entry details -- if necessary, but not required.

End of Idiom Discussion; on now to Non-Functional Requirements

XVI. **Non-functional requirements.**

A. So far in CSC 308 we’ve focused on functional requirements specification.

1. We are answering the question "*What* does the system do?"
2. This goes in Section 2 of the 308 requirements specification document.

B. There are also *non-functional* requirements to be specified.

1. For these we answering such questions as "How well", "How much?", and "How soon?" the system will perform, cost, and be delivered.
2. These requirements go in Section 3 of the 308 requirements specification document.

C. The non-functional requirements address aspects of the system other than the specific functions it performs.

D. These aspects include system performance, costs, and such general system characteristics as reliability, security, and portability.

E. The non-functional requirements also address aspects of the system development process and personnel.

F. Formally, a non-functional requirement is one that is not part of the program model.

G. There are three broad categories of non-functional requirements, elaborated on below: *System-related*, *Process-related*, and *Personnel-related*.

XVII. **Non-functional requirements by category.**

A. System-related non-functional requirements.

1. performance
   a. time
   b. space
2. operational environment
   a. hardware platform
   b. software platform
   c. external software interoperability
3. standards conformance
4. general characteristics
   a. reliability
   b. robustness
   c. accuracy of data
   d. correctness
   e. security
   f. privacy
   g. safety
   h. portability
   i. modifiability and extensibility
   j. simplicity versus power

B. Process-related non-functional requirements.

1. development time
2. development cost
3. software life cycle constraints

4. system delivery
    a. extent of deliverables
    b. deliverable formats
5. installation
    a. developer access to installed environment
    b. phase-in procedures to replace existing system
6. standards conformance
7. reporting
8. marketing
    a. pricing
    b. target customer base
9. contractual requirements and other legal issues

C. Personnel-related non-functional requirements

1. for developers:
    a. credentials
    b. applicable licensing, certification
2. for users:
    a. skill levels
    b. special accessibility needs
    c. training

XVIII. **The spectrum of preciseness and quantifiability in non-functional requirements.**

A. Some constraints are easy to state quantifiably -- e.g., "The system will respond within 10 seconds to a user request for local data, and within 60 seconds for a remote data request."

B. Some constraints and objectives are much harder to quantify -- e.g., the system will be "robust" and "user friendly"

C. In 308, we are formalizing only the functional, not the non-functional requirements.

XIX. **Cost/benefit/risk analysis.**

A. After a requirements specification is defined, it can be analyzed in terms of the costs, benefits, and risks involved in proceeding to develop the software.

B. Computer system analysts are not necessarily qualified to produce cost/benefit analyses alone; they typically need to consult with finance experts as well as the potential software design/implementation team.

C. The cost/benefit analysis can be included as a part of the requirements specification document or it can be a separate document.

D. In 308, we do not have time to address cost/benefit analysis.