CSC 308 Lecture Notes Week 10
Review of Milestone Deliverables
Modeling Idioms

I. **Milestone 8 -- see the handout.**

A. Due 11PM Tuesday 20 November.

B. The deliverables are:

1. Requirements updates
2. Prototype

II. **Milestone 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.

E. If necessary, add Late Updates appendix and 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. Some of these topics were discussed in the modeling reviews during Week 7 labs.

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 form of data.  The gist of
 authentication is that the login operation has a precondition that checks if
 a user's credentials are autthentic.  If so, the precondition succeeds ant the
 login operation returns the server's data.  If the user's credentials are not
 authentic, then the login precondition fails, and no data are returned.
 */
abstract class Server {
   Collection<UserRecord> users;
   Data data;

    /**
     * Login to this server to access its data.  If the user name and password
     * exist on the server's user list, then return the data, otherwise the
     * operation fails via precondition failure.
     */
    /*@
      requires
```

```
        \xists UserRecord user ; users.contains(user) ;
            user.id.equals(id) && user.password.equals(password);
      ensures \result.equals(data);

     @*/
    abstract Data login(String id, String password);
}

/**
 * A UserRecord is an ID/Password pair.  A more elaborate model could contain
 * other user information such as authentication level, e.g., admin level
 * versus plain user level.  Also, the model could specify security in the form
 * of password encryption operations.
 */
abstract class UserRecord {
    String id;
    String password;
}

/**
 * Data is a purely abstract class that represents the data stored on a server
 * that become 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 reason for indexing
 * from 1 is to correspond to the typical user-level understanding of a
 * sequence that's enuerated from 1, not from zero the way Java arrays and
 * lists are indexed.
 */
abstract class Sequence {
    List<Item> items;
    int cur_item;

    /**
     * Move to the next item in the sequence if not at the last item.
     */
    /*@
      requires cur_item < items.size();
      ensures cur_item == \old(cur_item) + 1;
     @*/
    abstract void next();

    /**
     * Move to the previous item in the sequence if not at the first item.
     */
    /*@
      requires cur_item > 1;
      ensures cur_item == \old(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 needec by a particular tool are modeled.
3. The full specification of HTML is refered to at a definintive 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, interspersed.  It's
 * important to note that this is a very abstract model of HTML that focuses on
 * the specific aspects needed by a particular project, NOT all of HTML.  As
 * necessary, aspects of this model can site the full HTML specification
 * defined by the W3C organization.  At the time of this writing, this spec is
 * at <a href= "http://www.w3.org/TR/html5/semantics.html".>
 * http://www.w3.org/TR/html5/semantics.html </a>.
 */
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.  If the tag
 * is TEXT, then the value holds a piece of plain text content that appears in
 * an HTML document.  If the tag is something other than TEXT, then it
 * represents an HTML command tag, of the form enclosed in angle brackets in an
 * HTML document.  For non-TEXT tags, the string value as the complete value of
 * the HTML command, including its arguments.
 */
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, i.e., tags that the tool uses explicitly.
 * For example, in EClass these include standard HTML list tags and others.  In
 * the TestTool and CSTutor, there may be other selected tags here, such as
 * IMG, if the tool provides a user command to insert images in the text of a
 * test question or tutorial page.
 *
 */
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 perfored in sequential steps.
 * To enforce this sequential behavior, each wizard step takes a unique type of
 * intput 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, or other operations can
```

```
 * produce the same type of object as an output, then adding an additional
 * component can make it unique.  In addition, a postcondition on stepNMinus1
 * can set the 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 step1.
     */
    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. **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.

```
/**
 * Class ToolWorkspace is simplified data model for a tool workspace that
 * contains the main data on which the tool operates, a copy of the data for
 * undoing, and a copy for redoing.  This model supports one-level undo/redo.
 * For multi-level undo, a list of undo data can be used.
 */
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.
    */
    /*@
      ensures
```

```
          undo_data.equals(\old(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.  In either case, the undo data
     * of the output workspace is nil, thereby allowing only one level of
     * undo/redo.  If undo data are used, then redo data are set to the
     * original data in the input workspace, thereby enabling the redo
     * operation.
     */
     /*@
       ensures
         undo_data == null
           ? data.equals(\old(data)) &&
             undo_data == null &&
             redo_data == null
           : data.equals(\old(undo_data)) &&
             undo_data == null &&
             redo_data.equals(\old(data));
      @*/
     abstract void undo();
}


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

End of Idiom Discussion; on now to Further Discussion of Overall Tool Modeling

X. **Evolution of Calendar Tool object model.**

A. The top-level Calendar Tool specification using modules.

1. Figure 1 is a UML packaging diagram for the modules in the Calendar Tool.
2. Your 308 projects can have similar modularization, as discussed in lab presentations and meetings.

B. Major Calendar Tool objects, first draft.

1. Figure 2 is a UML diagram for the first draft Calendar Tool object model
2. This model was derived from the initial requirements and represents the state of things around Milestone 6, when none of the formal pre- and postconditions have been formulated.
3. Also, the modularization is not yet well defined.

C. Major Calendar Tool objects, second draft.

1. Figure 3 is a UML diagram for the second draft Calendar Tool object model.
2. This model is a refinement of the first draft, based on the following activities:
   a. Adding a module structure based on the top-level command hierarchy, as shown in Figure 1.
   b. Discovering missing model details when doing operation specs, including pre- and postconditions.
   c. Discovering missing model details when refining the requirements scenarios; in particular:
3. The object model refinements discovered during operation refinement include the following:
   a. The need for a `PreviousState` component of `UserWorkSpace`, based on the specs for the `Undo` operation.
   b. The need for `Clipboard`, `Selection`, and `SelectionContext` components, based on the specs for the `Edit` module operations.
   c. The need for a `RequiresSaving` component in `UserCalendar`, based on specs for the `File` module operations.
   d. The need for the `RecurringId` component of `ScheduledItem`, based on specs for the operations

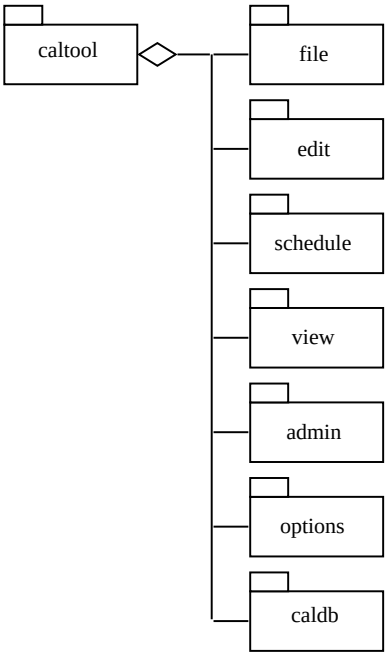

Figure 1: Calendar Tool Modules.

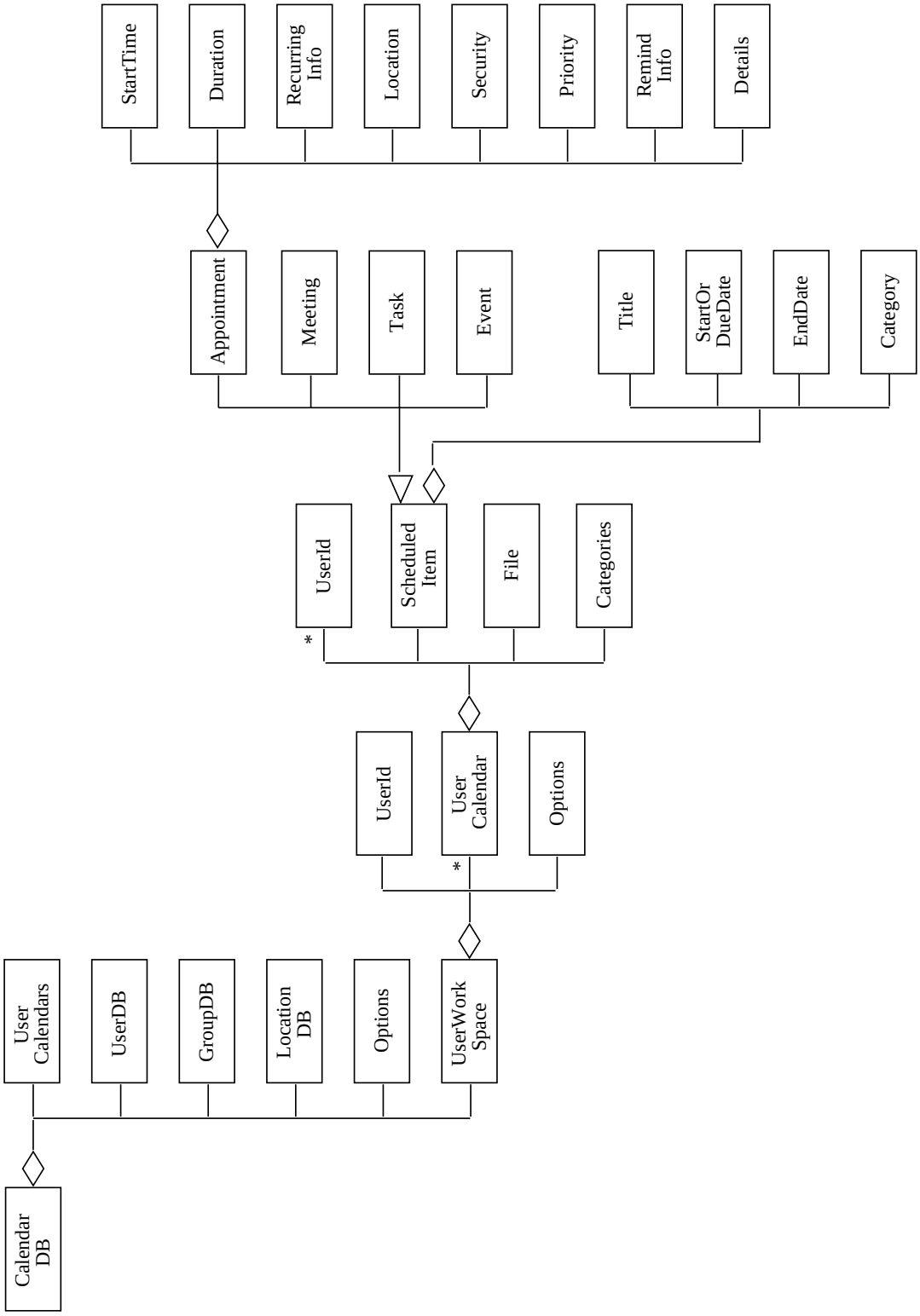

Figure 2: First-draft UML diagram for major Calendar Tool objects.

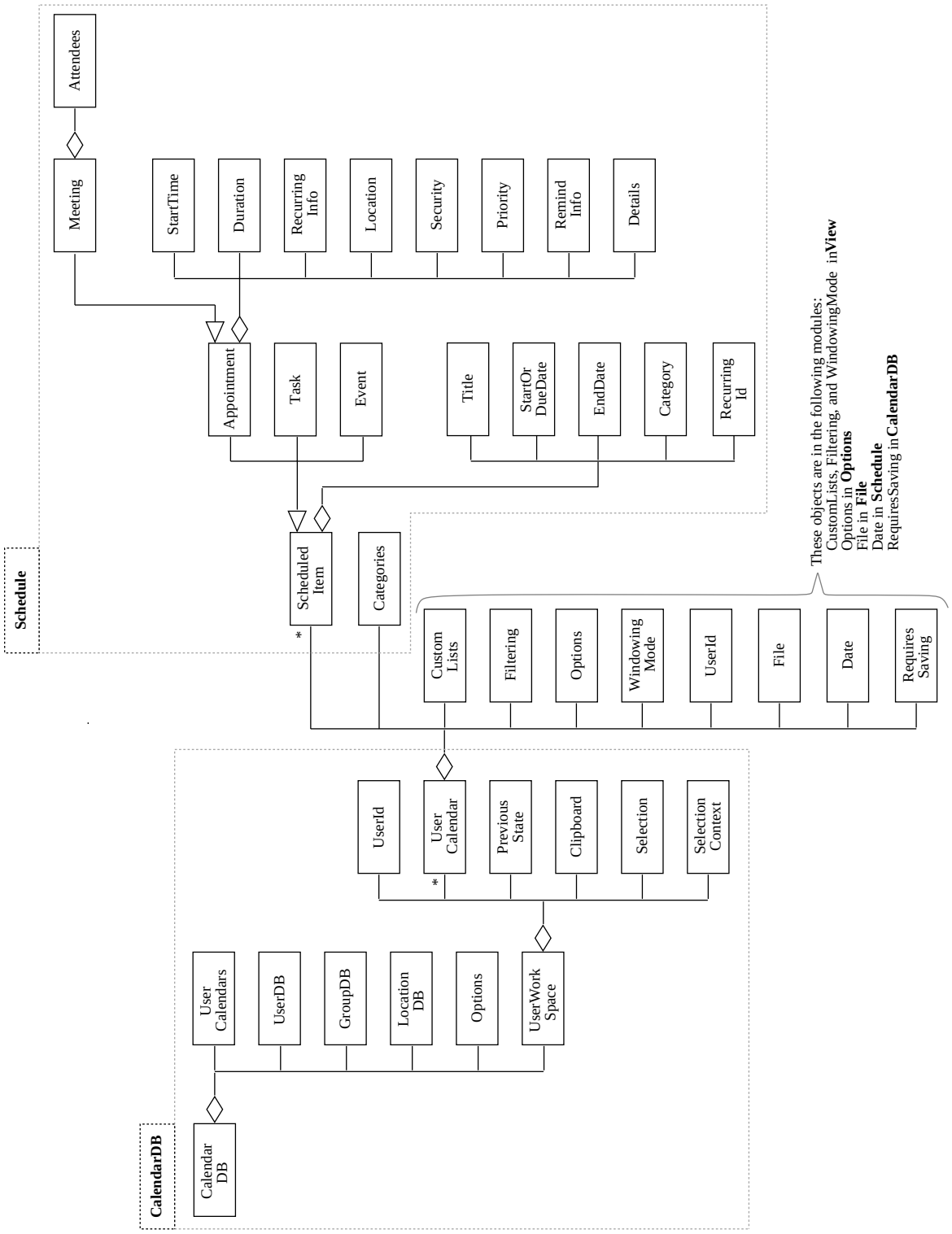

Figure 3: Refined UML diagram for major Calendar Tool objects.

that add and delete scheduled items.

e. Making Meeting a subclass of appointment.

4. The object model refinements discovered during operation refinement include the following:
   a. The need for `CustomLists` and `Filtering` components for `UserCalendar`, based on work done when refining the scenarios for the listing and filtering commands.
   b. Moving the `Options` component from the `UserWorkspace` to `UserCalendar`, based on refining the requirements to have calendar-specific options (rather than overall Calendar Tool options).

D. Specification details of the `CalendarDB` module, which contains the top-level Calendar Tool objects.

```
/****
 *
 * Module CalendarDB defines the major database objects of the calendar system
 * as well as the objects that define the underlying calendar structure.  The
 * core object of the calendar system is the UserCalendar, which is modeled as
 * a collection of scheduled items.  The calendar structure of years, months,
 * weeks, and days is not directly modeled in the UserCalendar.  These
 * structures are computed views that are derived from the underlying scheduled
 * item collection.  The operations in the View module are the ones that
 * compute the these views.
 *
 */

import java.util.Collection;

/**
 * The CalendarDB is the top-level object for the Calendar Tool.  The
 * first component is the list of UserCalendars for all registered users.
 * The next three components are the registered UserDB, the user GroupDB,
 * and the LocationDB.  The Options component is the set of calendar
 * options common by default for all users, both registered and
 * non-registered.  The UserWorkSpace component is active calendars upon
 * which the current user is working.
 *
 * objects that are available for the user to view and edit.  There is a <a
 * href="../images/uml.gif">draft UML diagram</a> of the Calendar Tool object
 * hierarchy, of which CalendarDB is the topmost object.
 */
abstract class CalendarDB {
    Collection<UserCalendar> calendars;
    UserDB udb;
    GroupDB gdb;
    GlobalOptions options;
}

/**
 * A user calendar is named, with its own set of options.  It extends the
 * Calendar object, which contains the collection of scheduled items.
 */
abstract class UserCalendar extends Calendar {
    String userName;
    UserOptions options;
}
```