CSC 307 Lecture Notes Week 4
Introduction to Requirements Modeling
Requirements Inspection Testing

I. This week's material:

A. Writeup for milestones 3 and 4

B. Milestone 4 example

C. Java as an Abstract Modeling Language

D. SOP Volume 2: Requirements Testing

II. Lab quiz Friday week 4, October 16th.

A. Covers material on SVN Basics handout.

B. Questions will be in terms of command-line interface to SVN.

C. There will be no questions on SVN clients.

D. It should less than 20 minutes to complete.

III. The next major phase of the software process -- requirements modeling and formal specification.

A. The goal is to formalize the user-oriented functional requirements, so that:

1. the requirements are complete and consistent;
2. the requirements are clear and unambiguous for the system design and implementation team.

B. While fully formal modeling of software is not (yet) practiced as widely as for other forms of engineered artifacts, the utility of formal software models is substantial. Semi-formal modeling is gaining wider acceptance in the SE world.

IV. When to model?

A. In a more traditional process, modeling is used a step in the successive refinemnt of software requirements into a concrete design and then implementation.

B. In a more agile / extreme programming processing, modeling is done as needed during a refactoring step.

C. Figure 1 is a comparative picture of these two approaches.

V. Languages to formally specify requirements.

A. Candidates include:

1. "Firmed up" English and pictures -- understandable but imprecise.
2. Semi-formal requirements specification languages -- helpful for high-level modeling, but not precise enough to ensure a complete and consistent specification.
3. Graphical notations -- helpful to clarify some aspects of a formal model, but not generally adequate for a complete specification.
4. A programming language or the abstracted version of a programming language.
5. Fully formal textual notations, including mathematics -- these remove all imprecision but are very demanding to use and understand.

B. Alas, "demanding to use and understand" is an attribute of many formal engineering notations.

1. Building and analyzing formal models is an important part of what engineers do to earn their keep.
2. Without a formal model, we run the very substantial risk of not fully understanding the system we want to build, and as a result building a faulty system.

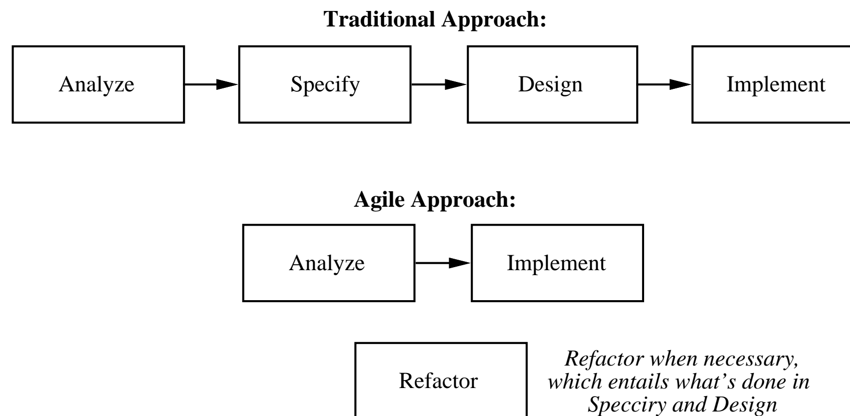


Figure 1: Where modeling fits in the process.

C. Why a formal language?

1. Remove the imprecision and ambiguity of normal English prose.
2. Avoid misunderstanding among analysts and potential users -- *consistency*.
3. Provide a means to identify when the requirements analysis process is finished -- *completeness.*
4. Provide some quantifiable measures by which to judge if a delivered system actually meets the requirements -- *verifiability.*

VI. Just how formal do we get?

A. In 307, we will go all the way down to formal mathematical logic.

B. We will do so in a sequence of steps from informal, to semi-formal, to fully formal.

C. Each step requires more work and more specialized knowledge.

D. In the real world, different participants in the analysis process will have different technical backgrounds.

1. Therefore, not all analysts will be involved with the most formal aspects of the document.
2. It is ultimately the job of the systems analyst to take input from all other analysts and produce a fully formal result.

E. Formality is particularly important in a growing number of "safety-critical" applications, such as avionics and medical systems, among others.

1. General rule -- the more important it is to prove that a computer system works properly, the more formally must it be specified.
2. Formal specification can be used in other areas that do not strictly involve safety, such as verifiably secure information processing system for financial transactions.

VII. Further details on formalizing the requirements.

A. The first step in formalizing user-oriented requirements is to build a *requirements model*.

B. The model is a more abstract representation of the requirements, written in a more formal language than English prose and pictures.

C. The objective in building the model is to depict the structure and operational behavior of a proposed system accurately and precisely.

D. Elements of the requirements model are the following:

1. The definition of *objects* upon which the system operates.

2. The definition of *operations* that the system performs.
3. The definition of object and operation *attributes*.
4. The definition of *relationships* between objects and operations.
5. *Statements of fact* about objects and operations, which statements can be validated to be true or false.
6. *Explanatory remarks* that aid in human understanding of the model.

E. The formal language we will use in 307 is Java, with modifications to make it suitable as an abstract specification language.

F. An overview is presented in the handout entitled "Java as an Abstract Modeling Language".

G. Here is a summary of using Java as abstract modeling language:

1. Objects are modeled as fully abstract Java classes or enums; no concrete classes, no interfaces.
2. Operations are modeled as method signatures; no method implementations.
3. Object attributes are modeled as Java annotations.
4. Object relationships are
   a. *has-a*, which is modeled as data fields
   b. *is-a*, which is modeled as inheritance using `extends`
5. Statements of fact are modeled with JML assertions (more on this in coming weeks)
6. The following Java features are *not used* in an abstract model:
   a. executable code
   b. information hiding with `public`, `private`, or `protected`
   c. exceptions
   d. library data structures except `Collection`
   e. primitive types except `int`, `double`, and `boolean`
   f. any other Java feature not explicitly mentioned above

VIII. Heuristics for deriving a requirements model from user-oriented requirements scenarios.

A. In our scenario style of requirements analysis, the requirements model is derived from the pictures of the user interface and the accompanying textual narrative.

B. The following heuristics can be used to derive an initial set of objects and operations from a graphical user interface:

1. Function buttons and menu items generally correspond to operations.
2. Data-entry screens and output screens generally correspond to objects.
3. More specifically, data-entry dialogs that appear in response to invoking an operation generally correspond to the input object(s) for the invoked operation.
4. Output reporting screens that appear in response to confirming an input dialog (E.g., with an "OK" button) generally correspond to the output object(s) for the confirmed operation.
5. Interface elements that allow entry of a single number, string, or boolean value correspond to primitive types.
6. The hierarchical structure of objects is generally displayed in the interface by nested or cascading windows and boxes, with primitive elements at the lowest level of nesting.

C. Specific details of object and operation attributes are derived from the scenario narrative that accompanies the interface pictures.

IX. Some examples from the Calendar Tool.

A. To illustrate the derivation of a requirements model, we'll apply the preceding basic heuristics to Calendar Tool example.

B. Complete details of the initial modeling for the Calendar Tool are in the specification directory of Milestone 4 example.

X. Deriving scheduling operations.

A. Here is the top-level `Schedule` command menu from the Calendar Tool:

```
Appointment ...
Meeting ...
Task ...
Event ...
```

B. Applying the first heuristic (buttons and menus indicate operations), we can identify the following four operations from the `Schedule` menu:

```
void scheduleAppointment();
void scheduleMeeting();
void scheduleTask();
void scheduleEvent();
```

C. We have not yet identified the following aspects of these operations:

1. What class they go in.
2. What parameter(s) they take.
3. What return value, if any, they produce.

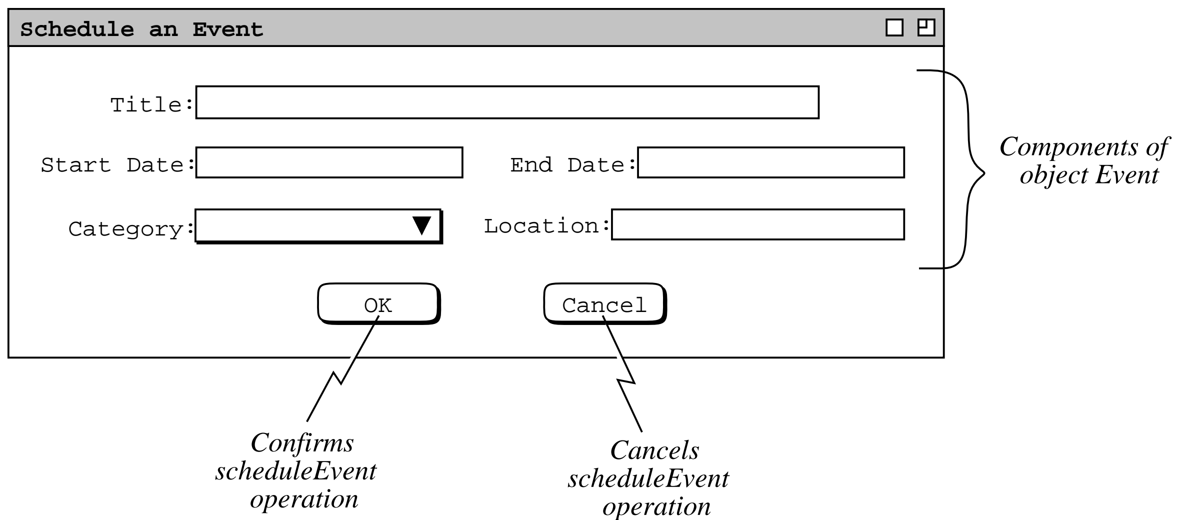
D. Linguistically, operation names should always be verbs or verb phrases.

1. Depending on how the user commands are structured, we can use different combinations of interface element names to derive meaningful operation names.
2. In this case, which is reasonably typical, we've concatenated a menu name with the name of each menu item to derive the operation names.
3. An important point is to have traceability between the terminology used in the user interface and the corresponding model.
   a. In fact, the derivation of model names can help point out flaws or inconsistencies in the interface scenarios.
   b. If it is difficult to derive a simple and meaningful name for an operation from the interface, this is a sign that the interface naming might well be improved.
   c. This is an instance of a recurring principle in requirements analysis and modeling -- "form follows function".
   d. That is, a well-defined interface scenario leads to a well-defined model, and vice versa.

XI. Deriving scheduling objects.

A. From the second heuristic (data screens are objects), we can identify as objects each of the data-entry screens that appear in response to the user selecting one of the `Schedule` menu operations.

B. To start with a simple example first, let us consider the derivation of the `Event` object, from the following interface picture:



C. Applying heuristics 5 and 6, we can derive the following object definitions:

```
class Event {
    String title;
    Date startDate
    Date endDate
    Category category
    String location;
}
class Date { /* ... */ }
class Category { /* ... */ }
```

D. In these definitions, we've done the following initial data analysis:

1. The title and location fields are primitive string type.
2. The other data fields are defined as object types that we've named, but not yet fully defined.

XII. Object derivation details.

A. As discussed in the "Java as Modeling Language" handout there are only a few Java forms used to model data

B. Table 1 summarizes these, along with the common interface forms.

C. These are constructs you should be familiar with in Java.

D. The table notes common interface forms for each of the basic object types.

XIII. Refining object definitions.

A. An examination of the narrative for the event dialog, indicates that the `Title` and `Location` components of an event are free-form strings, hence their definition as `String` types.

B. Java's `String` type is used to model a any free-form text string that the user may type.

C. In Milestone 4, the details of date formats has not yet been worked out.

1. Given this, we'll leave the definitions of the `Date` class to be resolved later.
2. I.e., we'll leave the definitions as

```
class Date { /* ... */ }
```

D. The user interface displays the `Category` as a list of selections.

1. This might lead us to consider modeling the `Category` component as a list of form

```
class Category { String* list; }
```

Java Form	**Meaning**	**Common Interface Form**
int	numeric integer	string editor for numbers; numeric slider bar or dial
double	numeric real number	same as integer
String	free-form string value	string editor or combo box
boolean	true/false value	string editor for true/false value; on/off button
class data fields	components of the class	box containing other types
enum literals	one of a set of possibilities	radio buttons; fixed-length listing of selections
`Collection`	zero or more components of the same type	variable-length listing of data values or selections
`Method`	the type of an operation	push button or menu item

Table 1: Java Modeling Forms.

2. However, a more careful analysis of the requirements shows that for a given event, the `Category` component is only one of a set of possibilities.
3. Hence, the `Category` component would not be a list, but rather just a primitive String.
4. Further analysis of the requirements shows that a category is not just a plain string, since each category has an explicitly selected color, as shown in the add-category dialog:

Add Category

Category Name:

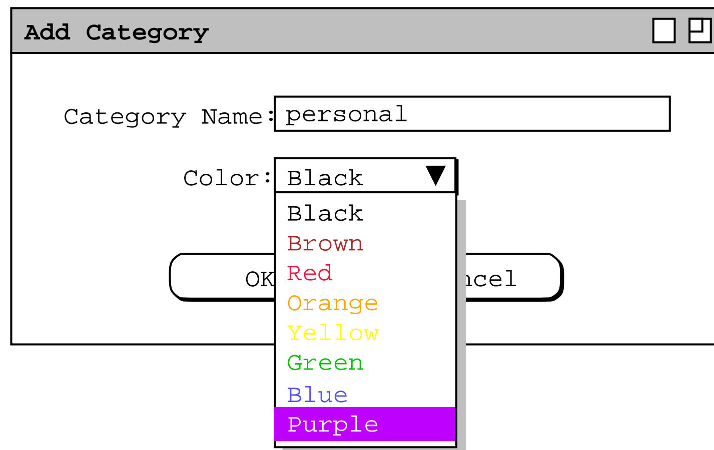
Color: Black ▼

OK Cancel

5. Hence, the most accurate definition of `Category` is

```
class Category {
    String name;
    Color color;
}
```

6. A subsequent screen shot in the scenarios shows that the `Color` component is one of a fixed set of selections:



7. Accordingly, we can model `Color` as follows:

```
enum Color {
    Black, Brown, Red, Orange, Yellow or Green, Blue, Purple;
}
```

E. The preceding analysis for deriving objects is typical in requirements modeling.

1. First we derive initial object definitions from the UI pictures.
2. Then we refine the definitions based on the scenario narrative.
3. We continue to refine until all objects are defined in terms of primitives, or we've decided to defer complete definition of model data until more requirements have been completed.

XIV. Refining operation definitions.

A. The key step in refining an operation is determining what object class it belongs in.

B. This will clarify what object is operated on.

C. In the case of the four scheduling operations, an analysis of the requirements leads us to understand that these operations work on a *Calendar* object.

D. Hence, we have the definition

```
class Calendar {
    void scheduleAppointment();
    void scheduleMeeting();
    void scheduleTask();
    void scheduleEvent();
}
```

E. Using heuristic 3 (data-entry dialogs are input objects), we refine the four scheduling operations as follows:

```
class Calendar {
    void scheduleAppointment(Appointment);
    void scheduleMeeting(Meeting);
    void scheduleTask(Task);
    void scheduleEvent(Event);
}
```

F. Since we want all of our models to compile with the Java compiler, we need to clarify that the preceding definition is intended to be an abstract model.

1. Abstract in this context means, among other things, that we leave out all operational code from the model.
2. Hence to compile in Java we must declare all of the methods to be `abstract`, as well as the class that contains these methods.

G. So, here is the compilable definition of the modeled Calendar object, along with its operations:

```
abstract class Calendar {
    abstract void scheduleAppointment(Appointment);
    abstract void scheduleMeeting(Meeting);
    abstract void scheduleTask(Task);
    abstract void scheduleEvent(Event);
}
```

XV. Identifying collection objects.

A. A key aspect of data modeling is the identification of *collection* objects.

B. Abstractly, a collection contains zero or more objects of a particular type.

C. In terms of requirements scenarios, collections can be identified by language that describes objects with multiple entries, and operations that add entries to the collection.

D. For example, in Section 2.2 of the Calendar Tool scenarios, the following kind of language helps identify the calendar as a collection of appointments:

> *"After scheduling and confirming an appointment, the appointment data are entered in an online working copy of the user's calendar."*

E. With Java as a modeling language, we will use the `Collection` interface to model abstract collections, as in this definition of `Calendar`:

```
abstract class Calendar {
    abstract void scheduleAppointment(Appointment);
    abstract void scheduleMeeting(Meeting);
    abstract void scheduleTask(Task);
    abstract void scheduleEvent(Event);

    Collection<Appointment> data;
}
```

F. Representing a Calendar as a collection of `Appointments` is in fact an over-simplification of a `Calendar`, since calendars can contain meetings, tasks and events, as well as appointments.

G. We'll address this issue soon, by defining a parent class for these four types of scheduled items, and representing `Calendar` data thusly:

```
Collection<ScheduledItem> data;
```

H. Another way to identify collections in requirements scenarios is by the pattern of operations that are used on collections.

1. The operations are *additive*, *destructive*, *modifying*, and *selective*.
2. In more common terms, these are operations to add, delete, edit, and find items in a collection.
3. In upcoming notes, we'll consider this to be a formal specification pattern.

XVI. Deriving a monthly view object.

A. A significant number of objects and operations will ultimately be derived from the calendar `View` commands.

B. As an initial example, consider in Figure 2 the monthly view that is displayed in response to the user selecting the `Month` item in the `View` menu.

C. From this we can derive the following objects:

```
import java.util.Collection;

/**
 * A MonthlyAgenda contains a small daily view for each day of the month,
 * organized in the fashion typical in paper calendars.
 */
class MonthlyAgenda {
```

Calendar Tool

File Edit Schedule View Admin Options Help

April 2015

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
28	27	29	30			

Figure 2: Monthly calendar view.

```
    FullMonthName name;
    DayOfTheWeek firstDay;
    int numberOfDays;
    Collection<SmallDayView> items;
}

class FullMonthName {
    String month;
    int year;
}

enum DayOfTheWeek { Sun, Mon, Tue, Wed, Thu, Fri, Sat }

/**
 * A SmallDayView has the number of the date and a list of zero or more short
 * item descriptions.
 */
class SmallDayView {
    int DateNumber;
    Collection<BriefItemDescription> items;
}

class BriefItemDescription {
    String title;
```

```
        Time startTime;
        Duration duration;
        Category category;
    }

    class Time { /* ... */ }
    class Duration { /* ... */ }
    class Category { /* ... */ }
```

XVII. Some observations on requirements modeling.

A. The Calendar Tool will provide some interesting examples where a model can be derived in a number of different ways.

1. For example, should the Calendar itself be modeled as a collection of scheduled items or as a collection of years?
2. Should dates be modeled as simple strings or a composite objects?
3. Which of these is the "correct" or "most accurate" way to model?

B. The general answer to such questions is that we strive to model objects and operations *as perceived by the end user*.

1. Our single criterion for model correctness and accuracy is based on how well we represent objects and operations in terms of what the user thinks.
2. What we definitely do not want to do is model things in terms of efficient computer data structures.
3. We will discuss these requirements modeling ideas more in upcoming lectures.

XVIII. Administrative matters.

A. Modeling for Milestone 4.

1. See the Milestone 4 example for roughly how much you should do.
   a. Each team member must commit at least four `Java` model classes, organized into packages.
   b. The model classes can be in one or more `.java` files.
   c. You'll need some team coordination for the major shared objects and the packaging structure.
2. Create package sub-directories in the project `specification` directory.
3. Put `.java` files to the appropriate package directories.
4. ***The files must compile with `javac`***.

B. Remember that this is the week of requirements inspection testing.

1. Review the procedure in the SOP Vol. 2 handout, which we'll go over in lecture on Monday.
2. In particular, be sure to decide in your team at what time on Friday (or even Thursday) pre-testing check-in is due in order for the inspection tester to get things done so the librarian can release them by 11PM.

XIX. Guidelines for modularizing a software model.

A. To *modularize* means to subdivide parts into independent units.

B. Here's an excerpt from the regular English dictionary definition for *"module"* that applies very well to software modeling:

> *"A module is an independent unit that can be used to construct a more complex structure"*.

C. In the specific case of a model defined in Java, modules are defined as `packages`.

D. A good heuristic for defining model packages is to use the large-grain structure defined in the software requirements

1. For example, each menu in a menu-based UI can be considered a module.
2. Similarly, the top-level UI toolbars can be considered to be modules.

E. Given these guidelines, the packaging structure of the Calendar Tool model can be defined as follows:

```
package file;
package edit;
package schedule;
package view;
package admin;
package options;
```

F. Within each package are the classes that support the package's functionality.

1. For 307 Milestone 4 example, the primary focus is on the `schedule` and `view` packages, since these contain the most important and interesting features of the Calendar Tool.
2. The packaging structure is easy to view in `javadoc` form in the Milestone 4 example.
3. Each package is documented with a file named "`package.html`" in each package directory.
    a. The file describes what the package is for.
    b. See the Milestone 4 example for what these files look like.

XX. Summary of core steps of the model derivation and refinement process.

A. **Derive initial model from UI screens**, using these heuristics:

1. Function buttons and menu items generally correspond to operations.
2. Data-entry screens and output screens generally correspond to objects.
3. More specifically, data-entry dialogs that appear in response to invoking an operation generally correspond to the input object(s) for the invoked operation.
4. Output reporting screens that appear in response to confirming an input dialog (e.g., with an "OK" button) generally correspond to the output object(s) for the confirmed operation.
5. Interface elements with a single number, string, or boolean value corresponding to primitive objects.
6. The hierarchical structure of objects is generally displayed in the interface by nested or cascading windows and boxes, with primitive elements at the lowest level of nesting.
7. Whole pull-down menus and large editing dialogs generally correspond to modules.

B. **Refine object model using requirements narrative.**

1. Define component details down to primitive-level objects.
2. Add inheritance based on references to "generic" objects mentioned in narrative.
3. Add object descriptions that synopsize narrative details.

C. **Refine operation model using requirements narrative and thoughtful functional analysis.**

1. Fully specify operation inputs and outputs.
2. Identify default inputs that the user does not need to enter explicitly in the interface.
3. Identify collection objects and add them to operation inputs/outputs, to ensure functional behavior.

XXI. Specific modeling guidelines.

A. Object and operation naming.

1. Derive object and operation names directly from requirements pictures and narrative.
2. The noun or noun phrase in the banner of a dialog is the name of the object derived from the dialog.
3. The labels of dialog components are the names of object components.
4. The verb or verb phrase on a menu item or function button is the name of the operation derived from the menu item or button.
5. Spaces and other alphanumeric punctuation must be consistently removed to form legal object name identifiers; otherwise, retain full spelling and capitalization in derived names, except for the Java convention to start method and data field names with a lower case letter.

B. Inheritance.

1. Derive inheritance relations based on explicit narrative in the requirements. The objective is to define inheritance in the model if it is perceptible in some form to the user.
2. Inheritance should not be used in a requirements model for the purposes of representational efficiency, as it is often used in programs.
3. The prime directive of modeling is "If the user perceives it, model it". Otherwise, leave it out.
4. In keeping with the prime directive, inheritance is generally best derived "bottom up", i.e.,
   a. Define all objects first without inheritance.
   b. Examine object definitions to see if there are common components.
   c. Define parent object classes based on common components.
   d. Confirm the use of inheritance in the model by finding justification for it in the requirements narrative (or adding such justification if the inheritance was discovered while modeling and is legitimately deemed "user perceptible").

XXII. Details of object derivation.

A. When interface screens are well laid out and clearly defined, object derivation is generally straight forward.

B. The following table summarizes the derivation of model types from common interface forms.

Common UI Form	**Model Object Type**
One-line Text box	Typically a string. If the requirements narrative defines specific constraints on what can be entered in the text box, then the model structure should reflect these constraints. E.g., if the narrative limits what can be typed to an integer value, then the text box is modeled as an integer. If the narrative defines what can be typed as a two-part value of some form, then the model is a two-tuple (e.g., Time and Date).
Multi-line Text Area	A string, list of strings, ore more complex object. A single string model is appropriate if the entire text area is entered as one large block of text the system does not decompose in any way. A list of strings is the appropriate model if the line-by-line contents of the text area are handled separately, but each line is not further decomposed. A more complex object model is necessary if the system performs any detailed parsing of the text to analyze its contents.
Fixed-length selection list	An `enum` object, with each element being one of the items in the selection list.
Variable-length list	A Collection object, with elements corresponding to the type of the selections.
Check-box	A boolean object. When check boxes are grouped together in the UI, model the group as a tuple of boolean objects.
Radio-button(s)	A boolean object or enum. A single on/off radio button is modeled as a boolean, e.g., a single radio button labeled "Yes/No" or "On/Off". When radio buttons are grouped together as a set of alternatives, the model is an enum with one component for each alternative. For example, with a group of radio buttons labeled "`Range`", with button labels "`High`", "`Medium`", and "`Low`", the model is `enum {High, Medium, Low}`
Dialog window	A class of objects that model the dialog components.

Tabbing or Multi-Panel dialog	A class with data fields for each tab panel, with each component class defining one of the tabs or panels.
Specialized "widgets"	Graphical interfaces may contain an assortment of primitive-level forms, such as on/off toggles, numeric slider bars, and small icons representing single values. A good rule for the use of such forms is if the corresponding model object is not readily derivable from the widget, then the widget is probably not that easy to understand for the user and should be replaced with a simpler form.

XXIII. Details of operation derivation.

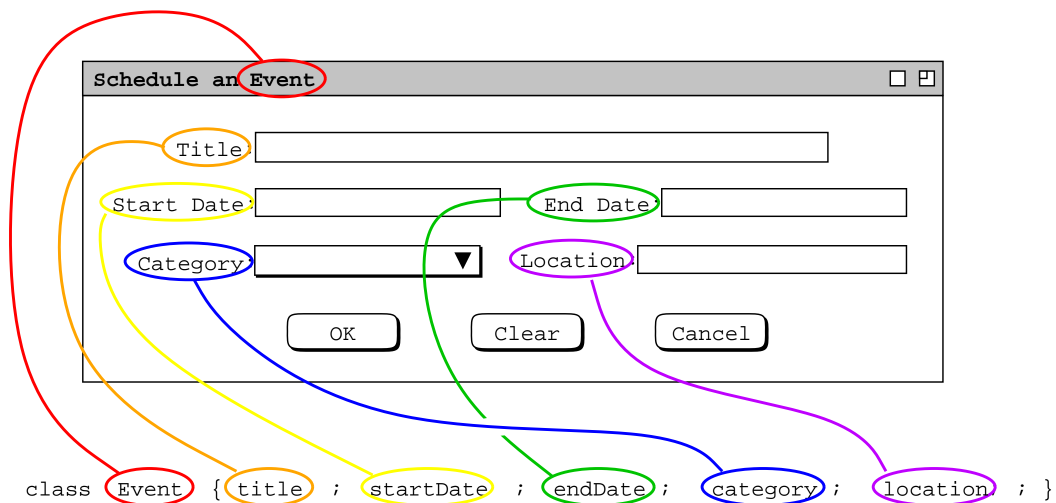
A. The "..." suffix in a menu item generally leads to two forms of dialog:

1. A data input dialog, with an `OK` button (or a button synonymous with OK).
   a. In this case, there is only one operation to model.
   b. Its name is derived from the menu item.
   c. The OK button itself is not a separate operation.
   d. Rather, there is three-phase GUI sequence to invoke a single underlying model operation:
      i. Select it from the menu (or function button).
      ii. Fill in the required values in the input dialog.
      iii. Confirm the operation
2. The alternative to a single-operation input dialog leading from "..." is a larger multi-operation dialog of some form.
   a. In this case, there are multiple operations to model, one each for the buttons or sub-menus in the dialog.
   b. The menu item itself does not derive an operation name, but rather a module name, in which the multiple dialog operations are defined.

B. No "..." in a menu item means that the input(s) required for the operation are collected as default values from the surrounding environment.

XXIV. Object and operation derivation examples from the Calendar Tool.

A. Event (introduced in Notes Week 4).



1. This annotated screen illustrates clearly the traceability between user-level requirements screens and underlying abstract model.
2. This is an admittedly simple example, however such traceability can be achieved throughout the requirements derivation process with some diligence.

B. Appointment

Schedule an Appointment

Title:

Date: Start Time:

hr min

End Date: Duration:

Recurring? Interval: weekly S M T W Th F S

Category: Security: public

Location: Priority: must

Remind? 15 minutes before on screen

Details:

OK Clear Cancel

```
class Appointment {
    String title;
    Date startDate;
    Date endDate;
    Time startTime;
    Duration duration;
    RecurringInfo recurringInfo;
    Category category;
    Location location;
    AppointmentSecurity meetingSecurity;
    AppointmentPriority priority;
    RemindInfo remindInfo;
    Text Details;
}
```

1. This illustrates a more involved dialog and its derived object.
2. Note the grouping of related UI components into the RecurringInfo and RemindInfo objects.
   a. This grouping in the model is a good cue that the UI itself could have better visual cues of how related components go together.
   b. E.g., the recurring and reminder areas could be surrounded by a box.
3. Also, the number of components violates the 7+/-2 rule.
   a. This is a cue that the dialog itself may be too complex for a single window.
   b. Some form of better ergonomic organization could be used, such as toggles to show more or less detail.

C. Meeting

Confirm a Meeting

Title:

Date: Start Time:

hr min

End Date: Duration:

Recurring? Interval: weekly S M T W Th F S

Category: Security: public

Location: Priority: must

Remind? 15 minutes before on screen

Attendees:

Details:

Minutes:

OK Clear Cancel

```
class Meeting {
    String title;
    Date startDate;
    Date endDate;
    Time startTime;
    Duration duration;
    RecurringInfo recurringInfo;
    Category category;
    Location location;
    AppointmentSecurity meetingSecurity;
    AppointmentPriority priority;
    RemindInfo remindInfo;
    Attendees attendees;
    Text Details;
    Text Minutes;
}
```

D. Task

Schedule a Task

Title:

Due Date:

End Date:

Recurring? Interval: weekly S M T W Th F S

Category: Security: public

Priority: must

Remind? 15 minutes before on screen

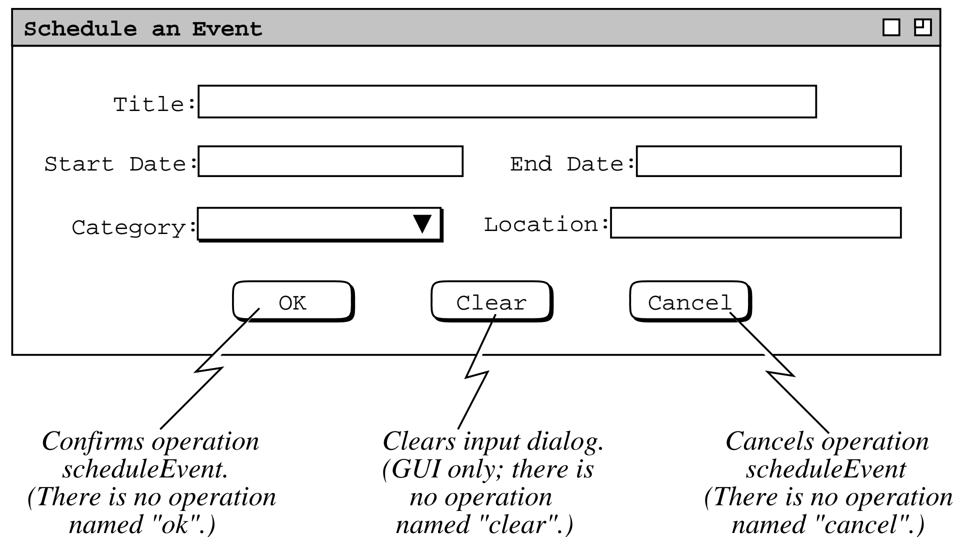
Details:

OK Clear Cancel

```
class Task {
    String title;
    Data dueDate;
    Date endDate;
    Category category;
    Security security;
    int Priority;
    RemindInfo remindInfo;
    Text details;
    boolean carryOverFlag;
    boolean completedFlag;
}
```

1. Note that there is a `CompletedFlag` in the model object that does not appear in the task scheduling dialog.
2. There is a to-do item in the Milestone 6 task-scheduling scenario that describes how the completed flag should appear once a task is scheduled.
3. This is an example of where the model is temporarily further developed than the requirements scenarios.

E. Deriving the `scheduleEvent` operation.




```
class Calendar {
     . . .
     void scheduleEvent(Event);
     . . .
}
```

1. As discussed in Lecture Notes 4, a major collection object has been identified -- the `Calendar`.
2. Each of the four scheduling operations is *additive* in that it takes a form of scheduled item and adds it to the Calendar.
3. An additive operation takes a collection component as input In an object-oriented language like Java,

   Hence, the functional form of the operation signature has the `Calendar` as both an input and output.

F. Refining the scheduling objects and operations.

1. Initial refinement of scheduled items using inheritance.

```
class ScheduledItem {
  title; startOrDueDate; endDate;  category; }

class Appointment extends ScheduledItem {...}

class Meeting extends ScheduledItem {...}

class Task extends ScheduledItem {...}

class Event extends ScheduledItem {...}
```

2. Second refinement pass, which adds component details, further refines inheritance, and refines operation signatures.

```
/*
 *
```

```
 * This file defines objects and operations related to calendar scheduling.
 * See Sections 2.2, 2.4, and 2.5 of the Milestone 6 requirements.
 *
 */

import java.util.Collection;

/**
 * The Calendar object is derived from an overall view of Sections 2.1 through
 * 2.5 of the requirements.  The functionality described in those sections
 * makes it clear that a Calendar is the primarily data object of the Calendar
 * Tool.
 *
 * The data component of a Calendar is a collection of scheduled items.  The
 * operations are those that schedule each of the four types of scheduled
 * item.  In the case of meetings, there are two operations involved -- one to
 * compute a list of possible times, and another to confirm a specific selected
 * meeting time.
 */
abstract class Calendar {

    Collection<ScheduledItem> data;

    /**
     * ScheduleAppointment adds the given Appointment to this.data, if an
     * appointment of the same time, duration, and title is not already
     * scheduled.
     */
    abstract void scheduleAppointment(Appointment appointment);

    /**
     * ScheduleMeeting uses the given MeetingRequest to determine possible
     * times that the requested meeting might be held, within the existing set
     * of scheduled items in the this.data.  The PossibleMeetingTimes output is
     * a list of zero or more possible times and dates that the meeting can be
     * held.
     */
    abstract PossibleMeetingTimes scheduleMeeting(
        MeetingRequest meetingRequest);

    /**
     * ConfirmMeeting takes a MeetingRequest, list of PossibleMeetingTimes, and
     * a Selected time from the list.  It adds a meeting to this.data,
     * comprised of the given request, scheduled at the selected time.  Further
     * details of output constraints are forthcoming.
     */
    abstract void confirmMeeting(
        MeetingRequest request,
        PossibleMeetingTimes times,
        int selectedTime);

    /**
     * ScheduleTask adds the given Task to this.data, if a task of the same
     * time, duration, and title is not already scheduled.
     */
    abstract void scheduleTask(Task task);

    /**
     * ScheduleEvent adds the given Event to this.data, if an event of the same
     * time, duration, and title is not already scheduled.
     */
    abstract void scheduleEvent(Event event);
```

```
}

/**
 * A ScheduledItem is the generic definition for the types of items stored in a
 * calendar.  The Title component is a brief description of what the item is
 * for.  The startOrDueDate and endDate components indicate when the item is
 * scheduled.  The category component is used to organize items into related
 * color-coded categories.
 *                                                                             <p>
 * There are four extensions of ScheduledItem.  They are Appointment, Meeting,
 * Task, and Event.  A ScheduledItem is derived from examining the common data
 * fields of these four types of item, and the requirements narrative that
 * describes these items.
 *                                                                             <p>
 * The startOrDueDate is a multi-purpose component of ScheduledItem.  Its
 * purpose depends on whether an item is a Task and whether it is recurring
 * (Events cannot recur).  For non-recurring appointments and meetings,
 * StartOrDueDate is used as the single date on which the item is scheduled.
 * If the item is recurring, StartOrDueDate is the first date on which it
 * occurs.  For a non-recurring Task, StartOrDueDate is the single date the
 * task is due.  If the task is recurring, StartOrDueDate is the first date it
 * is due.
 *                                                                             <p>
 * In recurring appointments, meetings, and tasks, the endDate defines the last
 * date on which the item will recur.  In events, the end date defines the last
 * date of a multi-day event.  When the value of end date is empty, the
 * startOrDueDate component is interpreted as the single date on which the item
 * occurs.
 */
abstract class ScheduledItem {
    String title;
    Date startOrDueDate;
    Date endDate;
    Category category;
}

/**
 * An Appointment adds a number of components to a generic ScheduledItem.  The
 * StartTime and Duration indicate when the appointment starts and how long it
 * lasts.  The Location is where it is held.  The Security indicates who can
 * see that the appointment is scheduled.  AppointmentPriority is how important
 * the appointment is.  RemindInfo indicates if and how the user is reminded of
 * the appointment.  Details are free form text describing any specific
 * appointment details.
 *                                                                             <p>
 * This object is derived from Section 2.2 of the Milestone 6 requirements, in
 * particular Figure 6.
 */
abstract class Appointment extends ScheduledItem {
    Time startTime;
    Duration duration;
    RecurringInfo recurringInfo;
    Location location;
    Security security;
    AppointmentPriority priority;
    RemindInfo remind;
    Text details;
}

/**
 * A Meeting adds two components to an Appointment.  The Attendees component
 * reflects the fact that a meeting is scheduled for more than one person,
```

```
 * whereas an appointment is for a single user.  The MeetingMinutes component
 * is a URL for the minutes of a meeting, once it has been held.
 *                                                                          <p>
 * This object is derived from Section 2.4.1 of the Milestone 6 requirements, in
 * particular Figure 46.
 */
abstract class Meeting extends Appointment {
    Attendees attendees;
    MeetingMinutes minutes;
}

/**
 * A meeting request has all the components of a meeting plus three additional
 * components to specify the latest dates and time at which the meeting can be
 * scheduled.  A meeting request is used to specify a range of possible meeting
 * times, to allow scheduling alternatives to be considered.  In the meeting
 * request, the inherited fields for startDate, endDate, and time are used for
 * the earliest dates and time at which the meeting can be held, i.e., for the
 * beginning values of each range.  The description of the ScheduleMeeting
 * operation has further details on how meeting requests are handled.
 *                                                                          <p>
 * This object is derived from Section 2.4.1 of the Milestone 6 requirements,
 * in particular Figure 45.
 */
abstract class MeetingRequest extends Meeting {
    Date latestStartDate;
    Date latestEndDate;
    Time latestStartTime;
}

/**
 * The PossibleMeetingTimes object is a collection of (start time, start date)
 * pairs at which a meeting could be held.
 */
abstract class PossibleMeetingTimes {
    Collection<TimeAndDate> timesAndDates;
}

/**
 * A TimeAndDate object is an element of a possible meeting time list.
 */
class TimeAndDate {
    Time startTime;
    Date startDate;
}

/**
 * Like an Appointment, a Task adds a number of components to a generic
 * ScheduledItem.  A Task differs from an Appointment as follows: (1)
 * Appointments have StartTime, Duration, and Location; Tasks do not.  (2) For
 * Appointments, the priority is either 'Must' or 'Optional'; for Tasks,
 * priority is a positive integer indicating the relative priority of a task
 * compared to other tasks.  (3) For appointments, reminders can be set to
 * occur at hour or minute granularity; for tasks, the smallest granularity of
 * reminder is a day.  (4) Tasks have a completedFlag, and completionDate
 * components; appointments do not.
 *                                                                          <p>
 * The completedFlag is true if a Task has been completed, false if not.  The
 * system does not enforce any specific constraints on the setting of a task's
 * CompletedFlag.  That is, the user may set or clear it at will.  Hence the
 * meaning of the completedFlag is up to user interpretation, particularly for
 * recurring tasks.
```

```
 *                                                                            <p>
 * The completionDate is the date on which as task is completed.  The system
 * does not enforce any specific constraints on the setting of a task's
 * completionDate (other than it being a legal Date value).  As with the
 * completedFlag, the meaning of the completionDate value is up to user
 * interpretation, particularly for recurring tasks.
 *                                                                            <p>
 * This object is derived from Section 2.4.2 of the Milestone 6 requirements,
 * in particular Figure 47.
 */
abstract class Task extends ScheduledItem {
    RecurringInfo recurringInfo;
    Security security;
    TaskPriority priority;
    TaskRemindInfo remind;
    Text details;
    boolean completedFlag;
    Date completionDate;
}

/**
 * An Event is the simplest type of ScheduledItem.  The only component it adds
 * to is Location.
 *                                                                            <p>
 * This object is derived from Section 2.4.3 of the Milestone 6 requirements,
 * in particular Figure 48.
 */
abstract class Event extends ScheduledItem {
    Location location;
}

/**
 * An AppointmentPriority indicates whether an appointment is a must or if it
 * is optional.  This information is used to indicate the general importance of
 * an appointment to the user.  The operational use of AppointmentPriority is
 * in the ScheduleMeeting operation, where the meeting scheduler can elect to
 * consider optional appointments as allowable times for a meeting.
 */
enum AppointmentPriority {
    Must,
    Optional
}

/**
 * A TaskPriority is a positive integer that defines the priority of one
 * task relative to others.  It's defined as a separate class in case we want
 * to enforce the value range restriction within the class constructor.
 */
abstract class TaskPriority {
    int value;
}

/**
 * For now, a Date is just as string.  This definition will expand soon.
 */
abstract class Date {
    String value;

    /**
     * Aux function used in scheduleEvent specs.
     */
    abstract boolean isValid();
```

```
}

/**
 * Duration is the time length of a scheduled item, in hours and minutes.
 */
abstract class Duration {
    int Hours;
    int Minutes;
}

/**
 * As with Date, Time is for now just as string.  This definition will expand
 * soon.
 */
abstract class Time {
    String value;
}

/**
 * RecurringInfo has components to specify the nature of a recurring item.  The
 * isRecurring component is an on/off flag that indicates whether an item
 * recurs.  The interval is one of Weekly, Biweekly, Monthly, or Yearly.  The
 * IntervalDetails component defines the precise means to define recurrence for
 * the different interval levels.
 */
abstract class RecurringInfo {
    boolean isRecurring;
    Interval interval;
    IntervalDetails details;
}

/**
 * Interval specifies the granularity at which recurring items are defined.
 * The Weekly and Biweekly settings allow the user to specify recurrence on one
 * or more days of the week.  The Monthly setting allows the user to specify
 * recurrence on one or more days in one or more weeks of each month.  The
 * Yearly setting allows the user to specify recurrence on one or more specific
 * dates in the year.
 */
enum Interval {
    Weekly, Biweekly, Monthly, Yearly
}

/**
 * IntervalDetails are either weekly or monthly.  This parent class is used
 * generically for either kind of details.
 */
abstract class IntervalDetails {}

/**
 * WeeklyDetails has an on/off setting for each day of the week on which
 * an item recurs.  These details are also used for the BiWeekly setting
 * of the recurrence interval.
 */
abstract class WeeklyDetails extends IntervalDetails {
    int onSun;
    int onMon;
    int onTue;
    int onWed;
    int onThu;
    int onFri;
    int onSat;
```

```
}

/**
 * MonthlyDetails can be specified on a day-of-the-week basis or on specific
 * date(s) basis.  The two extending classes have the specific details for these
 * two types of settings.  This parent class is used generically for either
 * kind of details.
 */
abstract class MonthlyDetails {}


/*
 * MonthlyDayDetails contains a weekly details component for each possible week
 * of a month.  The First- through ThirdWeekDetails are distinct for all
 * possible months.  Depending on the configuration of a particular month in a
 * particular year, there is potential conflict in specifying recurrence in the
 * fourth, fifth, or last weeks.  The conflicts are resolved as follows:
 *                                                                          <p>
 * For months with 4 weeks only, the settings in FifthWeekDetails do not apply,
 * and the settings in LastWeekDetails, if present, override any settings in
 * FourthWeekDetails.  For months with 5 weeks only, the settings in
 * LastWeekDetails, if present, override any settings in FifthWeekDetails.
 * (For months with 6 weeks, the LastWeekDetails component applies to the 6th
 * week, and there are no conflicts.)
 */
abstract class MonthlyDayDetails extends MonthlyDetails {
    WeeklyDetails firstWeekDetails;
    WeeklyDetails secondWeekDetails;
    WeeklyDetails thirdWeekDetails;
    WeeklyDetails fourthWeekDetails;
    WeeklyDetails fifthWeekDetails;
    WeeklyDetails lastWeekDetails;
}

/**
 * MonthlyDateDetails is a collection of zero or more specific dates in a month
 * on which an item recurs.
 */
abstract class MonthlyDateDetails extends MonthlyDetails {
    Collection<DateNumber> dates;
}

/**
 * A DateNumber is a positive integer between 1 and 31.  It's defined as a
 * separate class in case we want to enforce the value range restriction within
 * the class constructor.
 */
abstract class DateNumber {
    int value;
}

/**
 * A Category has a name and StandardColor, which serve distinguish it from
 * other categories.  Colored-coded categories serve visual cues to the user
 * when viewing lists of scheduled items in some form.  Categories can also be
 * used in filtered viewing.
 */
abstract class Category {
    String name;
    StandardColor color;
}
```

```
/**
 * A StandardColor is one of a fixed set of possibilities, per the requirements
 * scenarios.
 */
enum StandardColor {
   Black, Brown, Red, Orange, Yellow, Green, Blue, Purple
}

/**
 * For now a Location is a free-form string indicating in what physical
 * location an item is scheduled.  It may be refined to something like
 * (building,room) pair.
 */
abstract class Location {
    String value;
}

/**
 * Security is one of four possible levels, each of which is described
 * individually in the body of the enum.  The selected level specifies the
 * degree of visibility a scheduled item has to other users.  For an
 * appointment, task, or event, "other users" are defined as all users other
 * than the user on whose calendar the scheduled item appears.  For a meeting,
 * "other users" are defined as all users not on the Attendee list of the
 * meeting.
 */
enum Security {

    /**
     * Public security means other users can see the scheduled item and all the
     * information about the item.
     */
    Public,

    /*
     * PublicTitle security means other users can see the title of the
     * scheduled item but none of the other information about the item.
     */
    PublicTitle,

    /**
     * Confidential security means other users can only see that a user is
     * unavailable for the time period of a scheduled item; no other
     * information about the scheduled item is visible.  Since confidential
     * security applies to a specific time period, it is meaningful only for
     * appointments and meetings, not for tasks or events; tasks and events do
     * not have specific time components.
     */
    Confidential,

    /**
     * Private security means other users see no information at all about a
     * scheduled item, not even that the item is scheduled.  Note that private
     * security hides a scheduled item from the ScheduleMeeting operation,
     * q.v., so that a meeting may be scheduled at the same time as a private
     * appointment.  It is up to the user to handle this situation by
     * accepting or refusing the scheduled meeting.  Given the nature of
     * private security, it does not apply to meetings.  I.e., only
     * appointments can have private security.
     */
    Private
}
```

```
/**
 * RemindInfo has a flag that indicates if a scheduled item will have a
 * reminder sent and defines one of three ways that the user is alerted when a
 * scheduled event is to occur.  OnScreen means the user is reminded with a
 * pop-up alert on her computer screen.  BeepOnly means the user is reminded
 * with a simple audible tone on the computer.  Email means the user is sent an
 * electronic mail message reminder.
 */
abstract class RemindInfo {
    boolean isReminded;
    HowReminded howReminded;
}

enum HowReminded {
  OnScreen,
  BeepOnly,
  Email
}

/**
 * AppointmentRemindInfo extends RemindInfo by adding information for how
 * soon before a scheduled item the reminder is to be sent.  For appointments,
 * the time units are minutes, hours, or days (cf. TaskRemindInfo).
 */
abstract class AppointmentRemindInfo extends RemindInfo {
    double howSoonBefore;
    AppointmentReminderUnits units;
}

/**
 * TaskRemindInfo extends RemindInfo by adding information for how soon before
 * a task the reminder is to be sent.  For tasks, the time unit is days.  A
 * fractional day can be used for smaller granularity if desired.
 */
abstract class TaskRemindInfo extends RemindInfo {
    double howSoonBefore;
}

/**
 * Appointment reminders can come minutes, hours, or days before an
 * appointment.  The units for these can be fractional, for maximum
 * flexibility.
 */
enum AppointmentReminderUnits {
    MinutesBefore, HoursBefore, DaysBefore
}

/**
 * Attendees is a collection of names of those who attend a meeting.
 */
abstract class Attendees {
    Collection<String> names;
}

/**
 * MeetingMinutes is current defined as the URL for the location of the minutes
 * of a meeting.  This definition may be refined in upcoming versions of the
 * requirements.
 */
abstract class MeetingMinutes {
    String url;
}
```

```
/**
 * The details of the Text object are TBD.  It may just turn out to be a
 * plain string.  Or it may a limited form of HTML, so we can include linkable
 * URLs in it.
 */
abstract class Text {}
```

G. Observations.

1. Inheritance is generally easier to derive bottom up, than top down.
2. Remember -- "what the user thinks" is the driving factor in determining model accuracy and correctness.

XXV. Another example -- viewing objects and operations from the Calendar Tool.

A. Based on the Milestone 6 excerpt from Lecture Notes 3, the following shows some initial object derivation.

```
/*
 *
 * This file defines the objects and operations related to the different
 * calendar views available to the user.  See Section 2.3 of the Milestone 6
 * requirements.
 *
 * The structural viewing levels are item, day week, month, and year.  There
 * are operations to go to the previous and next views at any level, as well as
 * an operation to go to a specific date.  Lists of scheduled items can be
 * viewed in a variety of ways.  A general view filter operation can be applied
 * to both structural and list views.  Operations are available to view other
 * users' calendars and to view a list of active viewing windows.
 *
 * NOTE: this is work in progress.  A good deal of objects are yet to be
 * defined.
 *
 */
import java.util.Collection;

/**
 * A DailyAgenda has a full day name and a list of time-slot descriptors.  The
 * FullDayName consists of the day name itself (e.g., Wednesday), the month,
 * the date, and the year.  Each item in the TimeSlotDescriptor list consists
 * of a starting time (e.g., 8 AM) and a list of zero or more scheduled items.
 */
abstract class DailyAgenda {
    FullDayName name;
    Collection<TimeSlotDescriptor> times;
}

/**
 * A FullDayName has the complete and unique designation of a calendar day.
 */
abstract class FullDayName {
    DayName day;
    MonthName month;
    DateNumber date;
    YearNumber year;
}

enum DayName {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

enum MonthName {
    January, February, March, April, May, June,
```

```
    July, August, September, October, November, December
}

/**
 * A time slot descriptor represents one slot (physically, a row) in a daily
 * agenda.  The TimeSlotName component is the start time for the slot.  The
 * list of BriefItemDescriptors contains the items that begin within the slot,
 * where "within" is defined as the start time plus the current time increment.
 * The overlaps component is a list of items with start times that overlap with
 * an item in the BriefItemDescriptor list.
 */
abstract class TimeSlotDescriptor {
    TimeSlotName slotName;
    Collection<BriefItemDescriptor> itemDescriptors;
    Overlaps overlaps;
}

/**
 * A TimeSlotName consists of a numeric TimeValue and an AmOrPm indicator.
 * TO DO: this definition should be reconciled as appropriate with the
 * definition of Time in schedule.java.
 */
abstract class TimeSlotName {
    int timeValue;
    AMorPM amOrPm;
}

/**
 * A brief item descriptor contains a subset of the information for a full
 * scheduled item.  The information is a Title, StartTime, Duration, and
 * Category.
 */
abstract class BriefItemDescriptor {
    String title;
    Time startTime;
    Duration duration;
    Category category;
}

/**
 * Overlaps contain zero or more BriefItemDescriptors that overlap with with
 * the master item in a given time slot.  An overlapping item is one with a
 * start time within the same time slot as other items.  The "master" item in a
 * time slot is the item that is first in a sorted order based on start time,
 * duration, and alphabetic title as the primary, secondary, and tertiary sort
 * keys, respectively.
 */
abstract class Overlaps {
    Collection<BriefItemDescriptor> descriptors;
}

abstract class DailyFormatOptions {
    NormalTimeRangeOption normalTimeRangeOption;
    TimeIncrementOption timeIncrementOption;
    int incrementHeightOption;
    ShowOrHide showHide24HoursOption;
    ShowOrHide showHideExactTimeOption;
    ShowOrHide showHideDashedLinesOption;
    ShowOrHide showHideExtensionArrowsOption;
    OnOrOff proportionalSpacingOnOffOption;
    DisplayOverlapsOption displayOverlapsOption;
    DefaultHeightAndWidthOption defaultHeightAndWidthOption;
```

```
}

abstract class NormalTimeRangeOption {Time startTime; Time endTime;}
abstract class TimeIncrementOption {int hours; int minutes;}
enum ShowOrHide {Show, Hide}
enum OnOrOff {On, Off}
enum DisplayOverlapsOption {Horizontal, Vertical}
abstract class DefaultHeightAndWidthOption {int height; int Width;}


abstract class Hour {
    int value;   // Must be legal hour value
}

abstract class Minute {
    int value;   // Must be legal minute value
}

enum AMorPM { AM, PM }

abstract class WeeklyAgendaTable {
    FullWeekName name;
    Collection<WeeklyTimeSlot> slots;
}

abstract class FullWeekName {
    MonthName month;
    DateNumberRange dateRange;
    YearNumber year;
}

abstract class DateNumberRange {
    DateNumber start;
    DateNumber end;
}

abstract class WeeklyTimeSlot  {
    Collection<WeeklyItemDescription> items;
}

abstract class WeeklyItemDescription {
    DayName day;
    TimeRange range;
    String truncatedTitle;
}

abstract class TimeRange {
    Time start;
    Time end;
}

abstract class WeeklyAgendaList {
    FullWeekName name;
    Collection<DailyItemList> items;
}

abstract class DailyItemList {
    DayName name;
    DateNumber date;
    Collection<DailyItemDescription> items;
}
```

```
abstract class MonthlyAgenda {
    FullMonthName name;
    DayName firstDay;
    NumberOfDaysPerMonth numberOfDays;
    Collection<DailyItemDescription> items;
}

abstract class DailyItemDescription {
    String value;       // to be refined
}

abstract class FullMonthName {
    MonthName month;
    YearNumber year;
}

class NumberOfDaysPerMonth {
    int value;   // Must be between 28 and 31, inclusive
}

abstract class YearlyCalendar {
    YearNumber year;
    Collection<SmallMonthView> months;
}

abstract class SmallMonthView { /* ... */ }

/**
 * A YearNumber is a positive integer between 0 and 9999.  It's defined as a
 * separate class in case we want to enforce the value range restriction within
 * the class constructor.
 */
abstract class YearNumber {
    int value;
}

/*
 * Model operations to place in the appropriate class:
 *
DailyAgenda viewDay(Calendar);
WeeklyAgendaTable viewWeekTable(Calendar);
WeeklyAgendaList viewWeekList(Calendar);
MonthlyAgenda viewMonth(Calendar);
YearlyCalendar viewYear(Calendar);
 *
 */
```

B. Observations.

1. Models of the agenda at different levels have *all* necessary info, some of which may be filtered out from view based on option settings.
2. To get a complete model picture, all of the examples in the scenarios need to be examined.
   a. E.g., consider the components of object `TimeSlotDescriptor`:

      ```
      String timeSlotName;
      Collection<briefItemDescriptor> items;
      Collection<briefItemDescriptor> overlaps;
      ```

   b. Figure 10 (in the Milestone 6 requirements excerpt) shows the first two components.
   c. Figure 13 clarifies that the third component (`Overlaps`) is necessary.

C. Questions:
   1. Should the `DailyAgenda` object have a `DailyFormatOptions` component?
      a. Why or why not?
      b. If not, what object does have `DailyFormatOptions` as a component?
   2. Is there any reason to consider a parent class from which the different level agenda objects inherit?

XXVI. Summary observations about the analysis and specification phases of the software.

A. The goal of requirements modeling is to build an *abstract* model of the user-level requirements.
   1. Abstract means that certain details of the user-level description are left out.
   2. What is obviously left out is much of the English verbiage that is used to describe the system clearly in end-user terms.
   3. The other very important aspect of the abstraction is leaving out all *concrete UI details*, such as
      a. Buttons such as `OK`, `Clear`, and `Cancel` that are strictly GUI conveniences, not fundamental to the underlying model.
      b. Purely decorative aspects of the interface that make it "easy on the user eyes" but that do not represent fundamental properties of model objects or operations.

B. There is very beneficial feedback between the requirements analysis and specification phases of the software development process.
   1. Such feedback is a natural part of development since the user-level requirements, written in English prose and pictures, describe precisely the same system as the formal model, written in the formal specification language and graphical notations.
      a. The English requirements are understandable to human users and domain experts.
      b. The requirements model is understandable to the software analysts.
      c. It is very important that these two different representations are consistent with one another.
   2. This consistency is achieved by deriving the formal model from the user-level requirements, refining the model, and then transferring the refinements back to the user-level English and pictures.
   3. The "feedback loop" between English requirements and SpecL model specification continues until the user says the requirements are complete and the specification passes cleanly through the SpecL checker.

XXVII. Modeling the concrete GUI?

A. Are things like menus and dialog windows objects?
   1. The CSC 307 answer to these questions is "no".
   2. The reason is that we are defining *abstract* model specifications in which only data that are directly manipulatable by the user are modeled.
   3. The tool's concrete interface is not modeled as an object.

B. This modeling decision relates to the nature of the tool UIs we are specifying in 307, namely *direct manipulation* user interfaces.
   1. The term "direct-manipulation" describes the style of interface that gives the human user direct control over the functions performed by a software system.
   2. Modern WIMP interfaces (Windows, Icons, Menus, Pointing) are almost always direct manipulation in style.
   3. Direct manipulation UIs are in contrast to older style UIs in which the system had more control over when commands could be performed by the user.
   4. With a direct manipulation interface, we view the user as being in control of the operations that are performed, not the system.
      a. The end user may invoke any command directly via menus, with no explicit prompting from the system.
      b. The user may generally cancel commands at will.
      c. Conceptually, the system is an invisible part of what is going on.
   5. When modeling a system with a direct manipulation UI, we can abstract out objects that the user does not

change.

C. The UI structure of the tool does provide organizational guidance.

1. In particular, the hierarchical structure of the UI provides a good basis for the modular organization of objects and operations.
2. Hence, we define *modules* based on how the tool UI is organized.

D. Some observations about concrete UI modeling.

1. It is not *wrong* to model a GUI itself as an object.
2. In our case, we're following a convention to model only those objects that the user can change.
3. Hence for us, it is not *necessary* to model the unchangeable parts of the tool as objects.
4. There are cases where modeling a tool's UI is necessary.
   a. For example, some systems allow the user to do things like change the format of a toolbar, or define entirely new toolbars.
   b. In our 307 projects, we're not generally considering such tool features; if a 307 project does GUI building features, they need not be modeled formally.

XXVIII. Modeling the tool itself.

A. Is the Calendar Tool itself an object?, an operation?, a module?

B. There are a variety of ways to model the overall system itself.

C. One approach is not to model it at all.

1. In this approach, we completely abstract out the tool structure from the model.
2. This highly abstract view of the tool is consistent with the above convention to abstract out objects that the user does not change.
3. I.e., if the user cannot change the tool itself, it need not be modeled.

D. If we do choose to model the overall tool, it can be modeled as either an object or an operation, depending on the kind of processing that it performs.

1. There are two high-level models for an information processing tool such as we are building in 307 -- *transform-oriented* and *transaction-oriented*.
2. In a transform-oriented system, processing is viewed as transforming a single large piece of data from one form into another, using a single large operation.
   a. In a transform-oriented system, the inputs and outputs are typically large pieces of data that are widely different in structure.
   b. The transform operation takes the input, with some additional operational parameters, and transforms it into the output.
   c. A report-generation system is a good example of transform-oriented; it takes as inputs like a large database plus some format parameters, and produces a large report.
   d. A transform-oriented systems is best modeled at the top level as an operation.
3. A transaction-oriented system performs its work with a larger number of smaller operations, each one performing some form of incremental action.
   a. In transaction system, the difference between operation inputs and outputs is a typically small, incremental change.
   b. A database management system is an example of a transaction-oriented system, where operations to add, delete, and change database records make relatively small changes to the overall database.
   c. A transaction-oriented system is best modeled at the top level as an object.

E. In practice, most information processing systems are a hybrid of the two system types, comprised of both transformational and transactional components.

1. At the top-level, the CSC 307 projects are transaction-oriented.
2. There may be major operations within the systems that are transform-oriented.

F. As an example of top-level tool modeling, here is the outline for the Calendar Tool top-level module.

```
/****
 *
 * Class CalendarTool defines the top-level tool object that contains the
 * currently active calendar db, system state information, and an abstract file
 * space.
 *
 */
class CalendarTool {
    CalendarDB calendarDB;
    FileSpace fileSpace;
    SystemState systemState;
}

class CalendarDB { /* ... */ }
class FileSpace { /* ... */ }
class SystemState { /* ... */ }
```

G. We'll discuss top-level tool modeling further in upcoming lectures.

XXIX. Compiling an abstract Java model.

A. Use the standard `javac` compiler to check a model.

B. The examples shown above are unconventional in that they have multiple top-level classes in one file.

1. The `javac` compiler is OK with this.
2. As we refine the model, we will move to the typical Java convention of one class per `.java` file.

C. When we use Java's `Collection` interface, we must import at the top of the file with

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

This is the only import you'll need at the current abstract level of modeling.

D. A common error in early model development is to leave objects undefined.

1. The conventions in the 307 examples is to use this style for yet-to-be-defined objects

```
class Whatever { /* ... */ }
```

2. The comment with ellipses is a place holder indicating that there's more work to do.
3. Including the ellipses comment is a good practice, because the model will compile fine without them, but an undefined definition may be easier to overlook without some indication of its unfinished state.

E. You can use the standard `javadoc` documentation generator to produce a browsable version of the model.

1. It's a good idea to put `javadoc` output is a separate sub-directory of the model, so all of the generated files do not crowd the specification directory.
2. The convention used for the 307 examples is to have a `javadoc` sub-directory under the project `specification` directory.

XXX. Recap of testing in the software engineering process, and where requirements testing fits in.

A. In what might be called a traditional view of the software process, testing is seen as the last step, following implementation.

1. In this view, the program code itself is the only artifact that is subject to formal testing.
2. While code testing is critically important for quality software, the code is not the only artifact that should be tested.
3. In fact, all of the other major software process artifacts can be tested formally -- the requirements, the specification, and the design.

B. Figure 3 compares the position of testing as the final step of the process versus a pervasive step.

1. As discussed in Lecture Notes Week 1, pervasive steps run continuously throughout the development process, or at regularly-scheduled intervals.

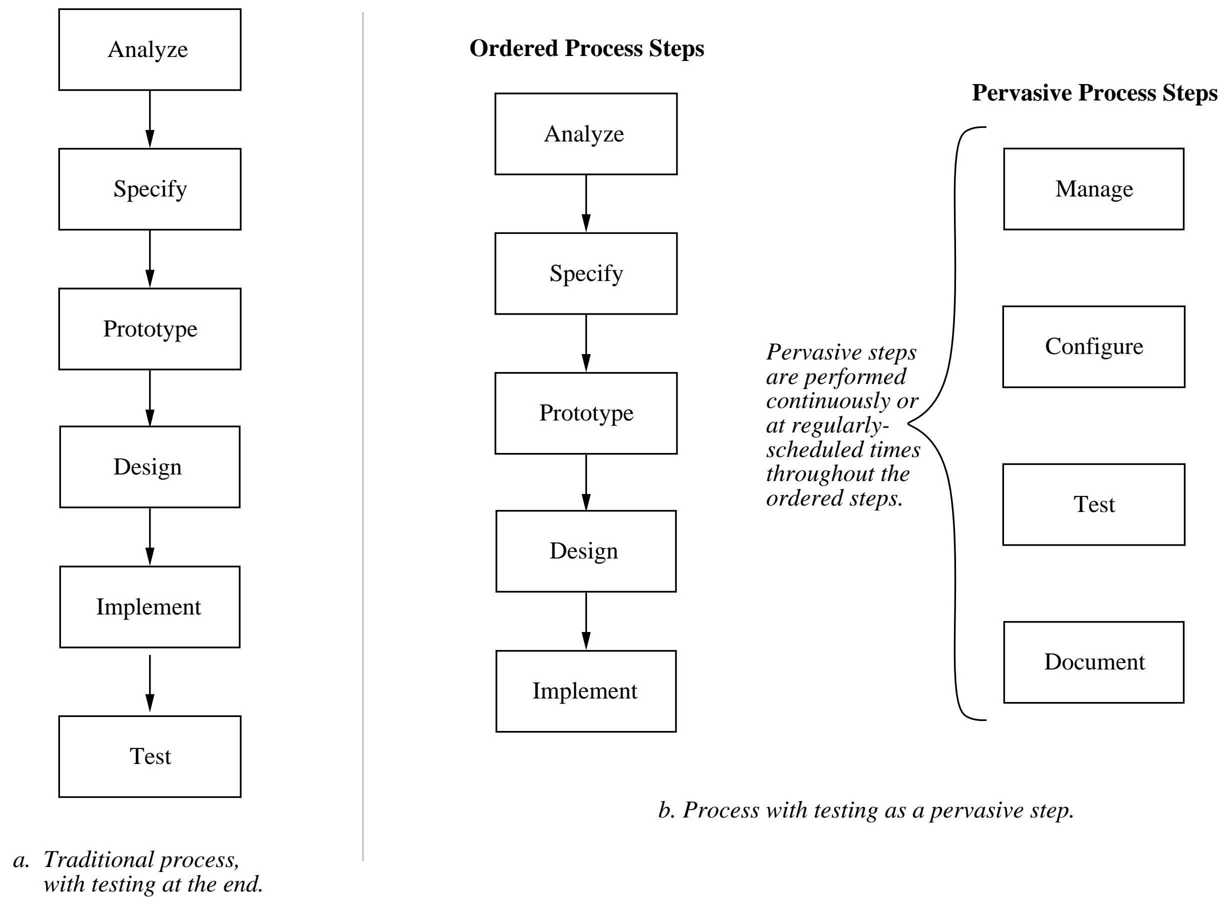


Figure 3: Two views of testing in the software process.

2. In addition to testing, the other pervasive steps deal with management, configuration, and documentation.

C. There are three types of testing that are performed during different stages of the software process.
1. *Inspection testing* entails systematic human inspection of all levels of software artifact, from requirements through implementation.
2. *Functional testing* is performed by programmers on the executable code as it is developed.
3. *Acceptance testing* is performed by end users on the released product.

XXXI. Inspection testing the requirements.

A. Testing with walkthroughs and reviews.
1. The purpose is the same as walkthroughs and reviews conducted during the development of just about any kind of product.
2. Namely, we want to assure that what is being developed is on track and meets customer needs.
3. Walkthroughs and reviews are an important means to "debug" the requirements.
4. Public reviews can be held at specific milestones during the course of requirements gathering and analysis.
5. Limited members of the technical staff hold detailed walkthroughs to refine requirements specifications.
6. Such walkthroughs are particularly important in the process of requirements analysis since such a wide range of people are potentially involved.
7. In 307, intra-group walkthroughs are conducted during our weekly meetings.

8. In addition, each group will gives two oral reviews to the rest of the class during in the quarter; first is in week 5 as scheduled above.

B. Formal inspection testing.

1. Starting in week 4, the functional requirements will be formally inspected by a duly appointed *inspection test engineer*.
2. During weeks 4 through 11, each group member will have a one-week assignment as the official inspection tester (see milestone 3 writeup for exact schedule, since it varies based on team size).
3. Details are in the handout entitled "Standard Operating Procedures, Volume 2: Requirements Testing"

C. Model building as a means of concept testing.

1. A common practice among engineers is to build a model of a proposed engineered artifact, to see if the high-level ideas about the artifact are sound.
2. For CSC 307, model building is done during the next ordered step of the software process after requirements analysis -- *specification*.