Asst Solution: Academic Appointment Calendar - Domain Model

Steven Zeil

Last Revision Date: February 22, 2008

1. Question 1: Candidate Classes and Responsibilities

For the initial list, mark up the description, looking for *noun phrases* and verb phrases .

We are considering the development of an *appointment calendar* specifically tailored to the academic environment in providing support for *faculty* *"office hours"* in addition to more conventional forms of *appointments* .

The *calendar* should provide features that are expected of typical existing calendar applications, such as

- The ability to record any number of *events* on one or more *days* . All events have a short *title* and may include additional *descriptive text* .
- Events may come in different forms. For example, "*Holidays* " apply to an entire day. "*Appointments* " occur at specific times within a day.
- The *owner* of a calendar may freely add , remove , and edit events , and should be able to look at events scheduled in a desired *month* , *week* , or *day* . The owner should have access to the calendar no matter at what *machine* he/she is located.
- Audible *alarms* may be associated with events to notify the calendar owner some *time in advance* of that event.
- Currently, many faculty post (on their *doors* or *bulletin boards*) a *list of* *"open" time slots* during which *students* and *other people* may schedule *meetings* simply by signing up for a *slot* . Most current online calendar software does not support this idea, and we are particulary interested that this new automated system should provide this capability.
- Some calendar events may repeat on a daily, weekly, or monthly basis. Calendar owner sometimes modify or remove individual *instances* of a *repeated event* (e.g., "we won't have our regular meeting this Monday because I'll be out of town") and sometimes need to modify or remove the entire *series* (e.g.,

"Let's start our regular Monday meetings a half hour earlier so we have more time before people have to leave.")

1) So the candidate classes are:

appointment calendar, faculty, office hours, appointments, calendar, events, days, title, descriptive text, Holidays, Appointments, owner, month, week, day, machine, alarms, time in advance, doors, bulletin boards, list of "open" time slots, meeting, time slot, repeated event, instance of repeated event, series of repeated event

Some of these may be synonyms (e.g., repeated event and entire series of repeated event). Some may appear to be synonyms, but questions that some teams put to the domain expert resulted in a clarification (e.g., Q: "Can anyone besides a faculty member be the owner of a calendar?" A: "Yes, advisors and other staff may find this useful as well. Even some students, especially graders and TAs, may need to schedule office hours in a similar manner.")

Some of these terms are also probably unimportant to the model (e.g., "doors, "bulletin boards"). Also omitted are terms that are clearly "meta" terms describing the problem statement or proposed project (e.g., "this new automated system").

2) In marking up the verb phrases, I ignored verbs that simply said one thing "is" another, or "is composed of" others. Those kinds of statements are capturing relationships between classes, which are important to the URL diagrams, but do not directly affect responsibilities except by suggesting attributes.

The candidate responsibilities are:

record events (in calendar), add event, remove event, edit events, look at events, associate alarm with event, notify owner of alarm, schedule meetings, sign up for a time slot, modify individual instance of repeated event, remove individual instance, modify entire series of repeated events, remove entire series of repeated events

Again, a few of these are synonyms (edit vs. modify, for example) and some may not be not particularly relevant, though relevance is usually best judged later as we begin to explore the interactions among the objects.

1.1. Grading Criteria

Completeness: Are all candidate classes and responsibilities included?

Some teams read that whole description and still only come up with 3-5 candidate classes. Many forgot to give a list of candidate responsibilities.

You can't argue that the 3-5 classes you chose are the only important ones. We won't know that for sure until much later in the analysis. That's why these are called *candidate* classes.

Correctness: Part of the OO approach is to always use terminology that is natural to the application domain. This facilitates communication both within the development team and between the developers and the domain experts and users.

If, for example, the description refers to "owners", don't call them "users, "people", etc. In some cases the terminology is unclear. People needed to ask the domain expert.

It's bad enough to invent terminology, but it's even worse to be so vague about it that no one could possibly know what you mean. "Weasel words" is my own term for words like "status", "information", "data", etc., that we are tempted to use as names to "weasel out" of the effort of coming up with a correct, precise term. E.g., when I see something like "get entry data", it's pretty clear that the author had no clue just what that data really was.

Defensibility: This is domain s. You are building a model of how the assessment world behaves. Don't introduce things that aren't mentioned and that therefore might not exist in that world.

In particular, note that the current world is described in terms of faculty using pieces of paper (sign-up sheets) posted in public places that students sign up on. Some items mentioned in the problem description (e.g., alarms) are part of a wish list for the software design, not part of the current world that you are supposed to be modeling.

A particularly bad form of unsupported classes are ones that are introduced just because people felt that the eventual program would need them. People who introduced classes for things like "standard file of University holidays", for example, were engaging in a late stage of design rather than an early stage of analysis. There's no mention of such a thing in the description. So we have no assurance that such a thing exists in the real world.

Such a file may, indeed, prove to be part of the implementing data structure for one of the classes. But this is analysis, and the focus is on modeling the real world.

2. Question 2: CRC cards

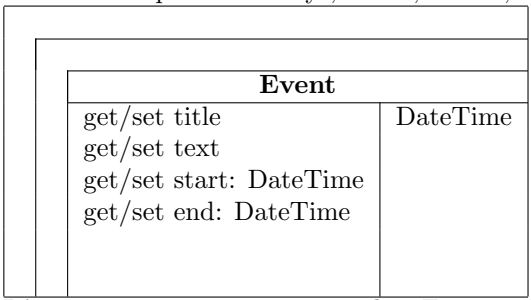
Calendar	
get/set owner	Owner
add/remove Event	Event
getEvents (time range)	EventSeries
add/remove EventSeries	

The "get/set" indicates that these are, in effect, attributes. "Add/remove" or, sometimes, "maintain" is used for attributes that are actually collections/plural – later analysis design will generally have to refine this to either introduce a named collection class as the attribute or to fill in the interface to indicate how individual items are accessed.

Some people chose to view the calendar as a container of days - not a bad model given the appearance of most wall calendars. I wouldn't deduct for this per se. But in response

to some teams' questions, I noted that events are often scrawled across multiple days on a paper calendar, which seems to suggest that the division of the calendar into days may not be consistent with or particularly important to the organization of events that seems to be the main focus of this description.

The "getEvents" function reflects the requirement that the owner should be able to view events stored in particular days, weeks, month, etc.



It's very tempting to presume that Event is the base class for an inheritance hierarchy with subclasses like holiday, appointment, meeting, etc. That certainly makes sense, although it's not clear how much variant behavior these subclasses will have. Some of the differences are just reflected in the start/end times (e.g., a holiday starts at midnight and ends one second before midnight on some day (possibly a different day, if you believe in multi-day holidays like "spring break"). However, meetings and appointments may include other info (e.g., a list of participants) that holidays would not have, so the idea of treating this as an inheritance hierarchy is certainly plausible.

Things start to get sticky, though, when we begin to consider repeated events.

EventSeries	
get/set title	DateTime
get/set text	
get/set start: DateTime	
get/set end: DateTime	
get/set repetition pattern	

RepeatedEvent	
get/set title	DateTime
get/set text	
get/set start: DateTime	
get/set end: DateTime	
get/set repetition pattern	

RepeatedEventInstance	
get/set title	DateTime
get/set text	
get/set start: DateTime	
get/set end: DateTime	

Is a repeated event the same thing as an "entire series of repeated events"? (probably). Is a repeated event an event? Is an instance of a repeated event an event? Can the answer to both of those questions be "yes", or are they mutually exclusive? If a repeated event is an event, do we mean to say that those are synonyms (all events have a capacity to be repeated) or is this an inheritance "is-a"? If repeated events are a subclass of event, how do we integrate this with the holiday/appointment/meeting hierarchy? Do we then need RepeatedHoliday, RepeatedAppointment, etc., classes?

This is legitimately tricky ground. The domain expert might be able to help (though no teams actually raised these questions), but it might also be a reflection of the natural tendency of natural language to include ambiguity.

Some plausible questions that might have been asked of the domain expert: "Q: When people enter repeated events into their calendars, do they enter them one at a time or all at once? A: They generally enter them all in one 'session', entering the earliest instance and then copying it to the next week, then to the week after that, and so on." "Q: Given a calendar marked up with repeated and non-repeated events, what do people use the terms

'appointment', 'meeting', etc. to mean - the single instance or the repeated series? A: Either one, depending on context."

The answer to the first question suggests that we might view an event series as a one-time "prototype" event (which might be an appointment, holiday, meeting, etc.) combined with a repetition pattern indicating how to produce later copies. The answer to the second indicates that the calendar should been seen, in some sense, as a container of both repeated and one-time events. So we get:

Calendar	
get/set owner	Owner
add/remove Event	Event
getEvents (time range): Events	EventInstance
getEventInstances (time range)	

Event	
get/set title	Eventinstance
get/set text	
getEventInstances (time range)	

EventSeries	
(subclass of Event)	EventIn
get/set first event instance	Repetit
get/set repetition pattern	

EventInstance	
(subclass of Event)	DateTime
get/set start: DateTime	
get/set end: DateTime	

Because both event series and instances are events, we only need a single add/remove events responsibility in the Calendar - not the separate addEvent and addEventSeries re-

sponsibilities that we listed before.

Note that the getEventInstances function for Events will have different implementations in the two subclasses. For EventSeries, this call could return any number of instances. For EventInstance, this would return zero or one instances (and if it returns one, it returns itself).

The rest of the cards are comparatively straightforward.

SignupSheet	
addTimeSlot	TimeSlot
getSlot (time range)	TimeRange
post	

TimeSlot	
get start	DateTime
get stop	NonOwner
sign up	
getPersonSigedUp	

Owner
Calendar
SignupSheet
Event
TimeSlot

Non-Owner
SignupSheet
TimeSlot

The only tricky part here is recognizing that the mention of Faculty and Students are really examples of Owners and Non-Owners, not definitional. The earlier-quoted question about who might own calendars is one clue to this. Another is the idea that open slots can be signed up for by "students and other people". Other people is so vague that it suggests that "students" is probably not, in itself, particularly significant.

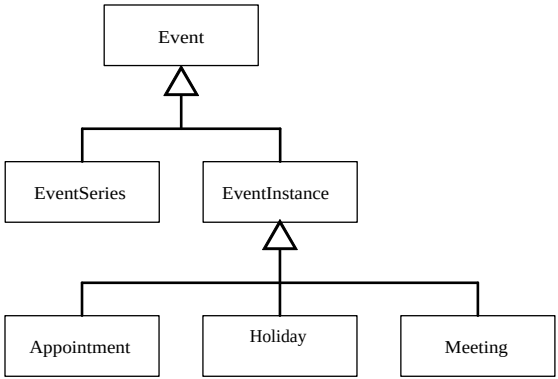
The last two cards illustrate a common pattern - that of an autonomous "actor" in the world who initiates events by issuing messages to other objects, but does not get calls from other objects.

These are the primary cards - the ones I would expect everyone to include.

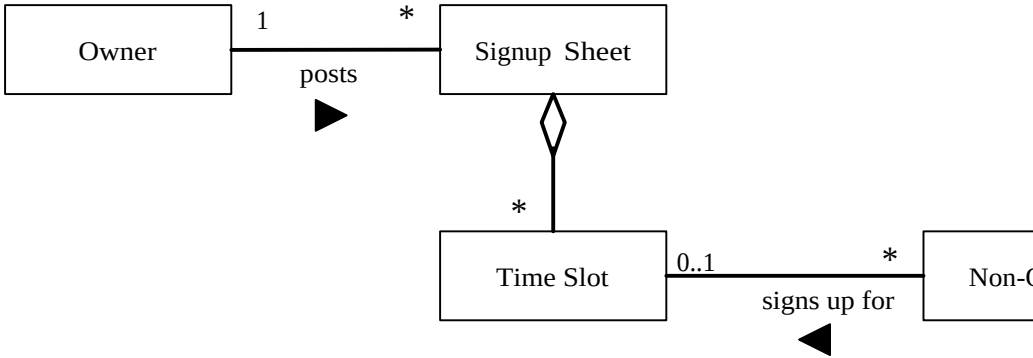
3. Question 3 - UML Class Relationship Diagrams

Clearly, there are many possibilities here. I'll illustrate just a few:

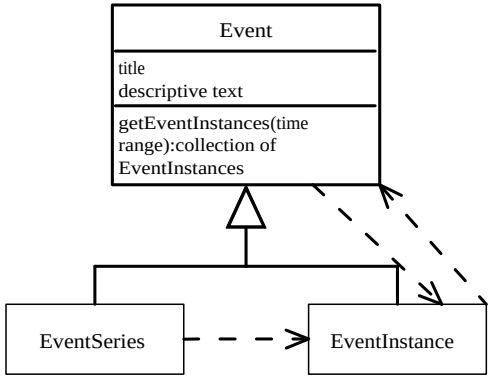
1. Calendar events may be single instances or series. Instances take many forms, including holidays, appointments, and meetings.



2. Many faculty post (on their doors or bulletin boards) a list of "open" time slots during which students and other people may schedule meetings simply by signing up for a slot.



3. It's a bit unusual to include an operation in the base class that returns elements of a subclass, as this sets up a circular dependency (every subclass is, obviously, dependent upon the interface of its base class). In an actual implementation, we would need to resolve this by returning pointers/references.



3.1. Grading Criteria

Notation: Are the diagram elements technically correct?

Completeness: Did students provide three diagrams with examples of three different kinds of associations? Are associations decorated with association/role names and cardinality when appropriate?

Correctness: Is the diagram an accurate portrayal of information from the model?

Defensibility: Is the information in the diagram justified by the available information?

Expressiveness: Does each diagram make a coherent statement about the model? Is there enough information provided for the diagram to convey useful information?