A Meeting Room Scheduling Problem

Objective Engineering, Inc.

699 Windsong Trail
Austin, Texas 78746
512-328-9658
FAX: 512-328-9661
ooinfo@oeng.com
http://www.oeng.com

© Objective Engineering, Inc., 1999-2007.

Photocopying, electronic distribution, or foreign-language translation of this document is permitted for personal and classroom use, provided this document is reproduced in its entirety and accompanied by this notice and by its copyright. Copies and translations may not be used or distributed for profit or commercial advantage without prior written approval from Objective Engineering, Inc.

Part I: The Problem

Design a system to schedule meetings and meeting rooms. A user can use this system simply to request a room of a given size for a given period of time. For example, a user can request a room that will hold 30 people from 1 p.m. until 3 p.m. this Friday. In addition, a user can request that an existing meeting (already defined in the system with a set of attendees) be scheduled at with a particular starting time and ending time. For example, a user can ask to have the Browser Project staff meeting scheduled this Thursday from 2 p.m. until 3 p.m. (That meeting has already been defined in the system and currently includes 11 attendees.)

A user can cancel any scheduled meeting or any room assignment up until the point at which the meeting or assignment begins (i.e., up until 1 p.m. on Friday and 2 p.m. on Thursday in the above two examples, respectively).

When a meeting is scheduled, an electronic message about that meeting must be sent to each attendee. Likewise, when a meeting is canceled, each attendee must be informed by electronic mail about the cancellation.

A user must also be able to define or alter a meeting. When defining the meeting, the user provides a list of attendees. The user may alter a meeting definition by adding attendees to or removing attendees from the meeting. A user may also remove an entire meeting definition. Note that adding or removing attendees has no effect on scheduled instances of that meeting (unless the last attendee is removed from a meeting, in which case future scheduled occurrences of that meeting should be canceled). A result of removing a meeting, on the other hand, is that all scheduled instances of that meeting must be canceled.

Assume the existence of a Post Office package that contains Post Office and Address classes. The Post Office class defines one method:

`deliverMessage(recipient : Address, message : String)`. It delivers the specified message to the specified recipient. (Assume that all messages are delivered.)

Assume the existence of an Employee Management package that defines an employee management component. That package exports a facade class, Employee Management, and an Employee interface class. The facade class defines the following methods:

`employee(employeeNumber : integer) : Employee`. Given an employee number, this method returns as type Employee a reference to an object defining the employee with that employee number.

© 1999-2007 by Objective Engineering, Inc.

The Employee interface defines the (abstract) methods:

- `address( ) : Address`. This method returns the electronic mail address of the employee.
- `name( ) : String`. This method returns the name of the employee.

© 1999-2007 by Objective Engineering, Inc.

Part II: A Solution

The meeting scheduling problem describes a system that schedules meetings and rooms. The solution to this problem presented here consists of a requirements model and an initial design.

The requirements model includes a use case diagram using standard UML notation. To augment that diagram, the model also contains a description of each use case. Although UML prescribes no particular form for such descriptions, this model specifies each use case in terms of its trigger, preconditions, and postconditions.

The design is cast as a class diagram and a set of interaction diagrams. Additional textual descriptions of several classes are included.

While this document describes a relatively simple meeting scheduling system, the reader may wish to ponder various extensions. For example, the system could:

- Allow a user to request that a meeting be scheduled at the earliest time possible (or perhaps at the earliest time on or after some specified date).
- Consult a personal calendar for each attendee when scheduling a meeting. An employee's calendar indicates when that individual is not otherwise engaged. A meeting can be scheduled only if all attendees are available.
- Allow employees to be grouped (such as by project), and designate groups (as well as individuals) as attendees of a meeting.
- Maintain two lists of attendees for a meeting: those who must attend versus those whose attendance is optional.
- Allow a user to obtain a list of his or her meetings.

A Requirements Model

The meeting scheduling system has at least one type of actor: the Scheduler. Despite its name, this role is played by a person who requests that a meeting or room be scheduled. Recall, however, that the system permits its users to define meetings and to alter those definitions. Can the persons who schedule meetings and rooms also define new meetings? If not, the requirements model must include another actor modeling a second role, Meeting Administrator. Note, however, that these actors define *roles*; in some cases, the same individual may play both roles.

A person playing the role of a Scheduler can perform the following functions:

a) Schedule a particular meeting for a particular time interval;

b) Request a room for a particular time interval;

c) Cancel a particular meeting scheduled for a particular time; and

d) Cancel a room assignment for a particular time.

The Schedule Meeting, Schedule Room, Cancel Meeting, and Release Room use cases, respectively, will be used to model these four functions. A person acting as a Meeting Administrator can do the following:

a) Define a meeting;

b) Add an attendee to a meeting;

c) Remove an attendee from a meeting; and

d) Remove a meeting definition.

These will be modeled by the Define Meeting, Add Attendee, Remove Attendee, and Remove Meeting use cases, respectively.

Two of these use cases *always use* (or include) two others as a part of their behavior. Scheduling a meeting requires the use of scheduling a room to select a room for the meeting. Likewise, canceling a meeting employs the behavior for releasing a room. These appear as `«uses»` (in UML 1.1 and 1.2) or `«includes»` (in UML 1.3) relationships in a use case diagram.

What happens when a Meeting Administrator removes a meeting definition from the system, but scheduled instances of that meeting exist in the future? In that situation, those scheduled instances should be canceled. One approach to modeling that relationship is to indicate that Cancel Meeting extends (i.e., `«extends»`) Remove Meeting. That is, the Remove Meeting behavior is conditionally extended by the behavior of the Cancel Meeting use case.

Likewise, if the last attendee is removed from a meeting definition, and future scheduled instances of that meeting exist, those instances should be canceled (thereby releasing the room). This implies that the Cancel Meeting use case `«extends»` Remove Attendee.

Because a Post Office instance and the Employee Management component are external software applications with which the meeting scheduling system must communicate, you should include those two entities as actors. Their interactions with the scheduling system include:

- When defining a meeting, a Meeting Administrator specifies a list of attendees in the form of employee identifiers. (An employee identifier is any unique key for an employee, such as an employee number or system user name). If the scheduling system must check those identifiers for

validity, it must interact with the Employee Management component. (For this solution, however, assume that the employee identifiers are validated before they are presented to the scheduling system.)

- Scheduling and canceling a meeting requires an interaction with the Post Office, as each attendee must be informed. To obtain each attendee's electronic mail address, those two use cases must also interact with the Employee Management component.

Figures 1a and 1b contain UML 1.1/1.2 and UML 1.3 use case diagrams, respectively, for this problem. Each includes the actors, use cases, and use case relationships described above.

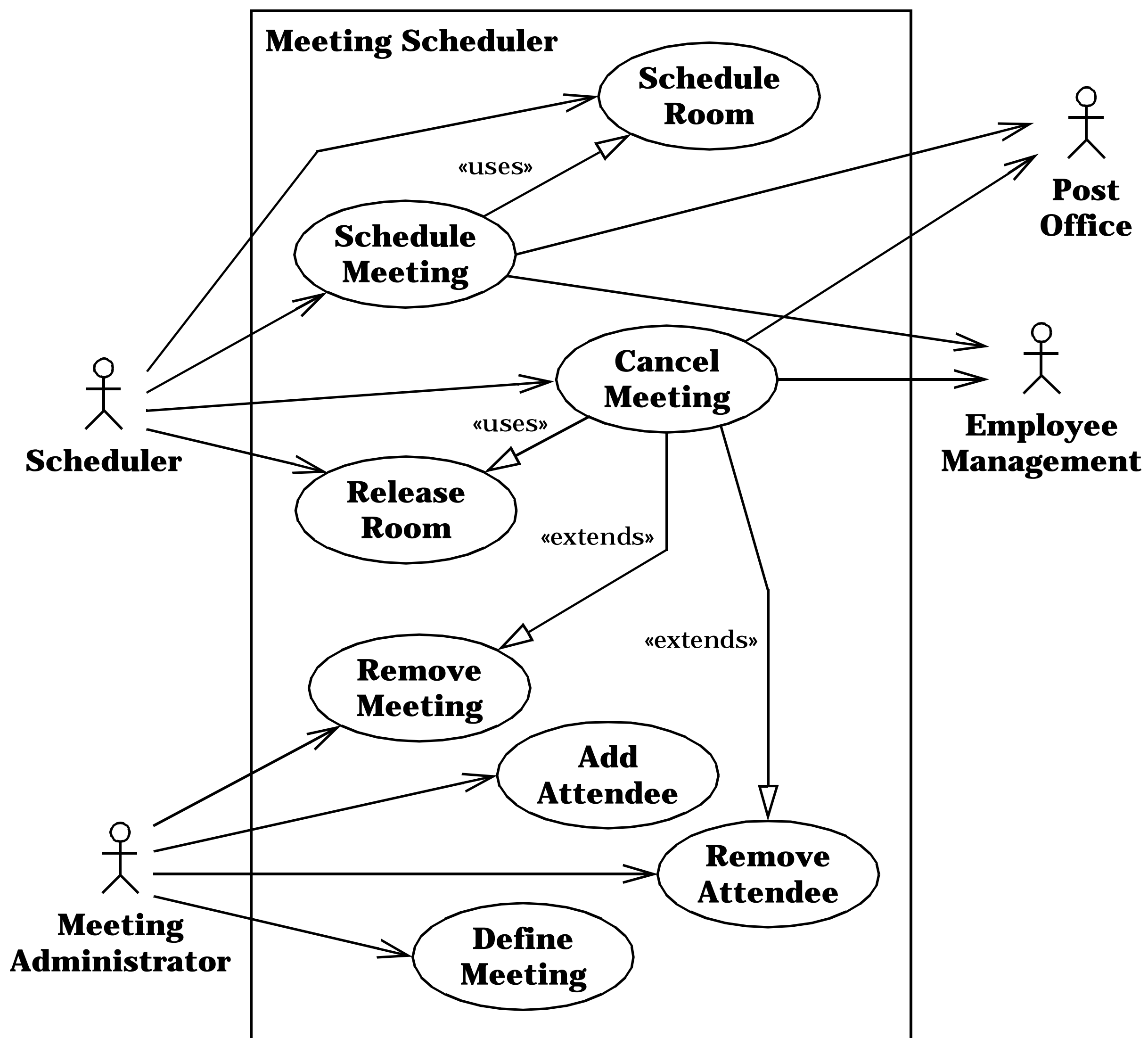

Figure 1a: A UML 1.1/1.2 use case diagram for meeting scheduling.

© 1999-2007 by Objective Engineering, Inc.

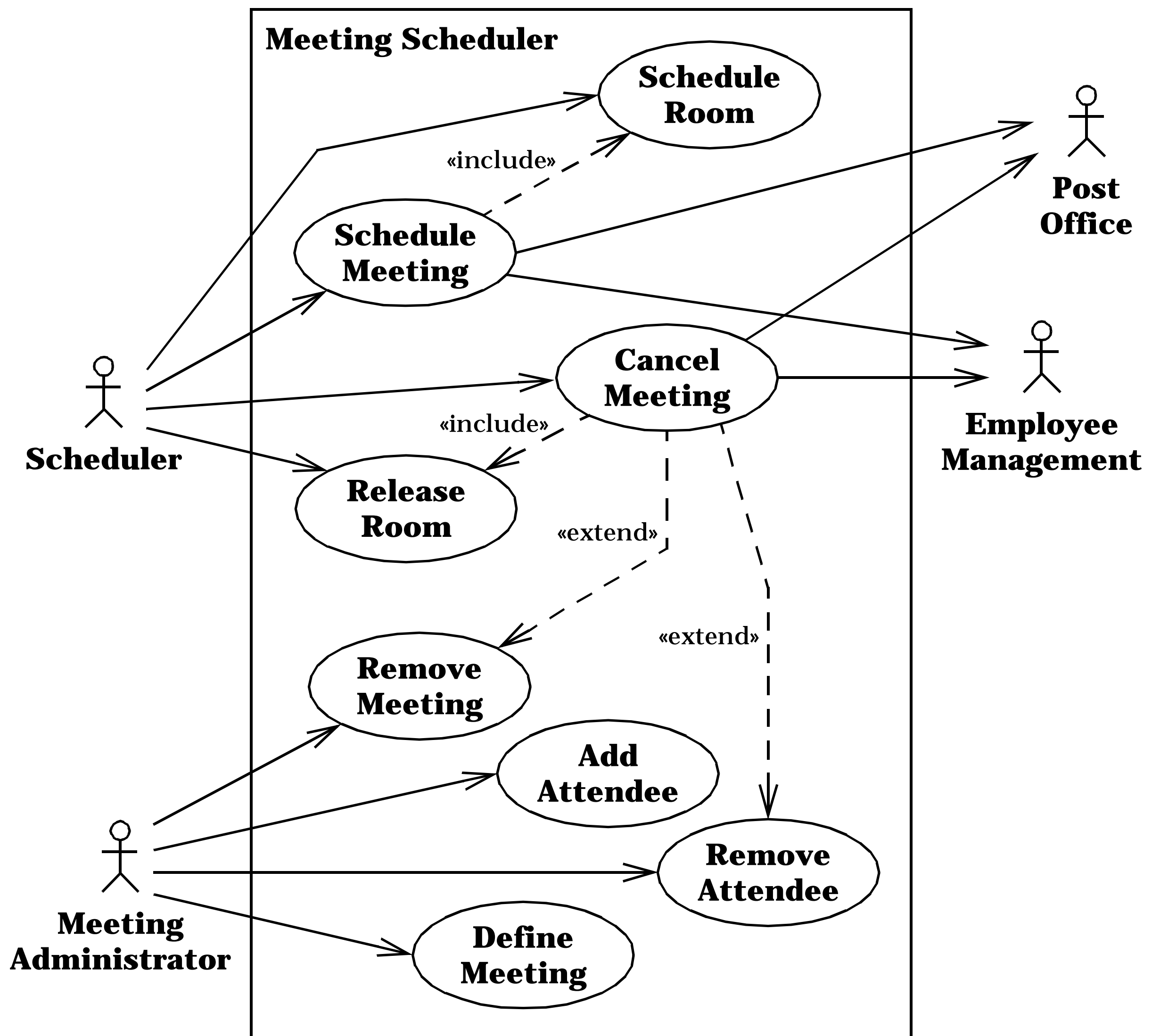

Figure 1b: A UML 1.3 use case diagram for meeting scheduling.

The use case diagram in Figure 1 ignores some exceptional behavior that may arise during the use of the scheduling system. One exceptional condition occurs when a room is requested but no room of the given size is available for the given period. If handling this condition requires non-trivial behavior, such as logging the failed request for a room, that behavior could be modeled as a separate use case (e.g., Diagnose Unavailable Room) that extends Schedule Room. Because it entails only returning an error to the user, however, it is wrapped into the Schedule Room use case in this solution.

© 1999-2007 by Objective Engineering, Inc.

At least two other exceptional situations result from adding an attendee to or removing an attendee from the definition of a meeting. What if an attendee is added to a meeting definition of which there are scheduled meetings in the future? It seems reasonable to assume that the attendee is expected to be present at those meetings and therefore must be informed of their dates. Likewise, when an attendee is removed from a meeting definition of which there are future scheduled meetings, the attendee should be informed that his or her attendance at those meetings is not required.

Consider Mary's staff meeting, a specific meeting definition. An existing attendee of Mary's staff meeting must be informed when Mary's staff is scheduled (by the Schedule Meeting use case) to meet next Monday at 10 a.m. An attendee subsequently added to the definition of Mary's staff meeting (by Add Attendee) must also be informed of next Monday's meeting. Do these two acts constitute the same behavior?

If the two behaviors are essentially identical, as is assumed here, then a single use case, Inform of Meeting, can model that function. That use case is used by Schedule Meeting (because it *always* informs its attendees), and it extends the Add Attendee use (because it occurs only in cases where a future meeting is scheduled).

Likewise, an Inform of Cancellation use case is used to inform the employee that the employee should not attend a specified meeting at a specified time, either because the meeting has been canceled, or because the employee is no longer an attendee of the meeting. The Cancel Meeting use case uses the Inform of Cancellation use case, which in turn extends the Remove Attendee use case.

The use case diagrams in Figures 2a and 2b are each an elaboration of one of the previous use case diagram. In particular, it includes the Inform of Meeting and Inform of Cancellation use cases and their relationships. Observe that those new use cases interact with the Employee Management and Post Office actors (to obtain electronic mail addresses and to deliver messages, respectively).

One way to describe the use cases in Figures 2a and 2b is by specifying each use case's trigger, preconditions, and postconditions. (Note that this form is *not* defined as a part of UML.) The Define Meeting use case is invoked to add a meeting definition to the system. It has the following specification:

Trigger:

- an event,
  `defineMeeting(name, attendees: array of employeeId)`

Preconditions:

- A meeting named `name` does not already exist in the system.

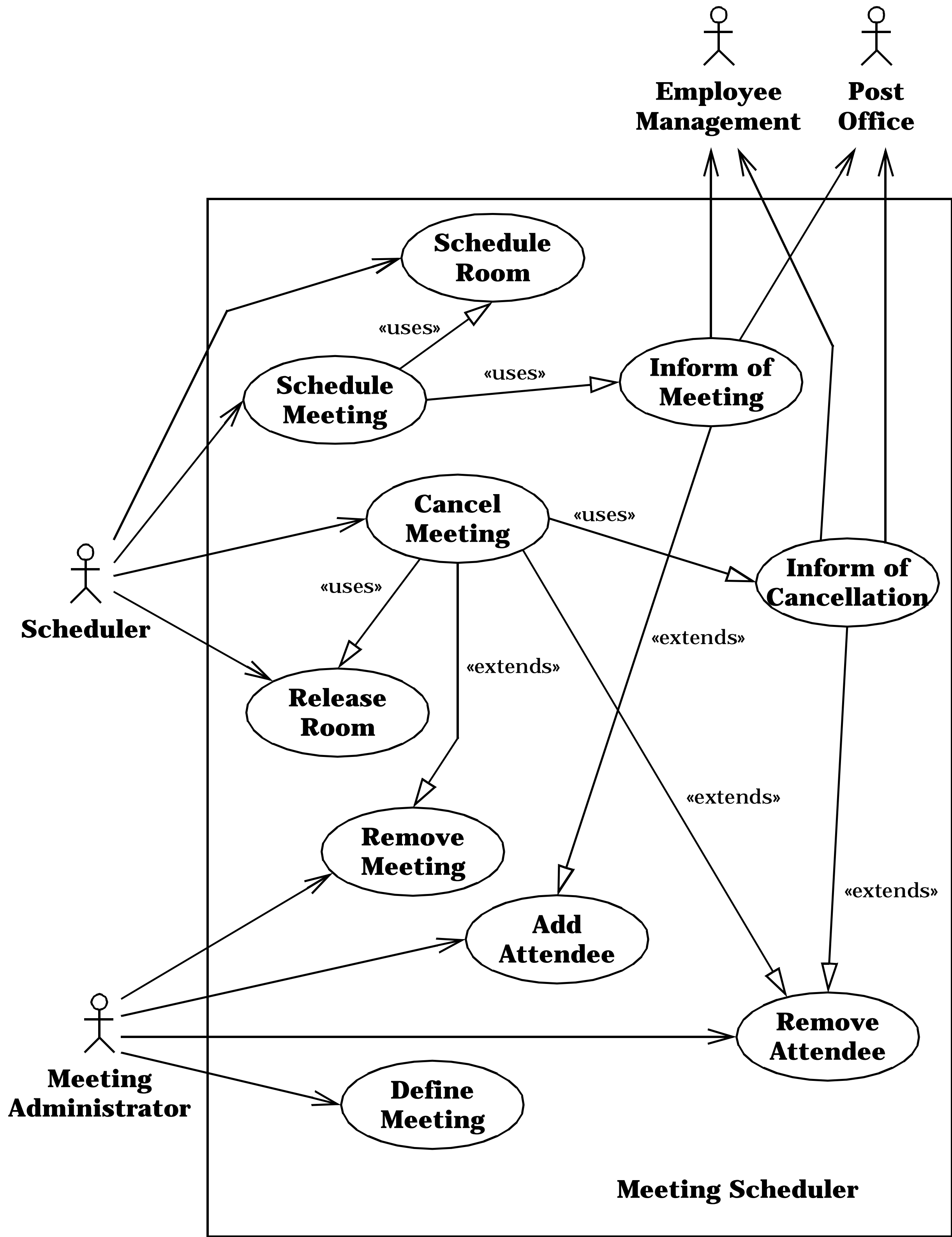

Figure 2a: A more detailed (UML 1.1/1.2) use case diagram.

© 1999-2007 by Objective Engineering, Inc.

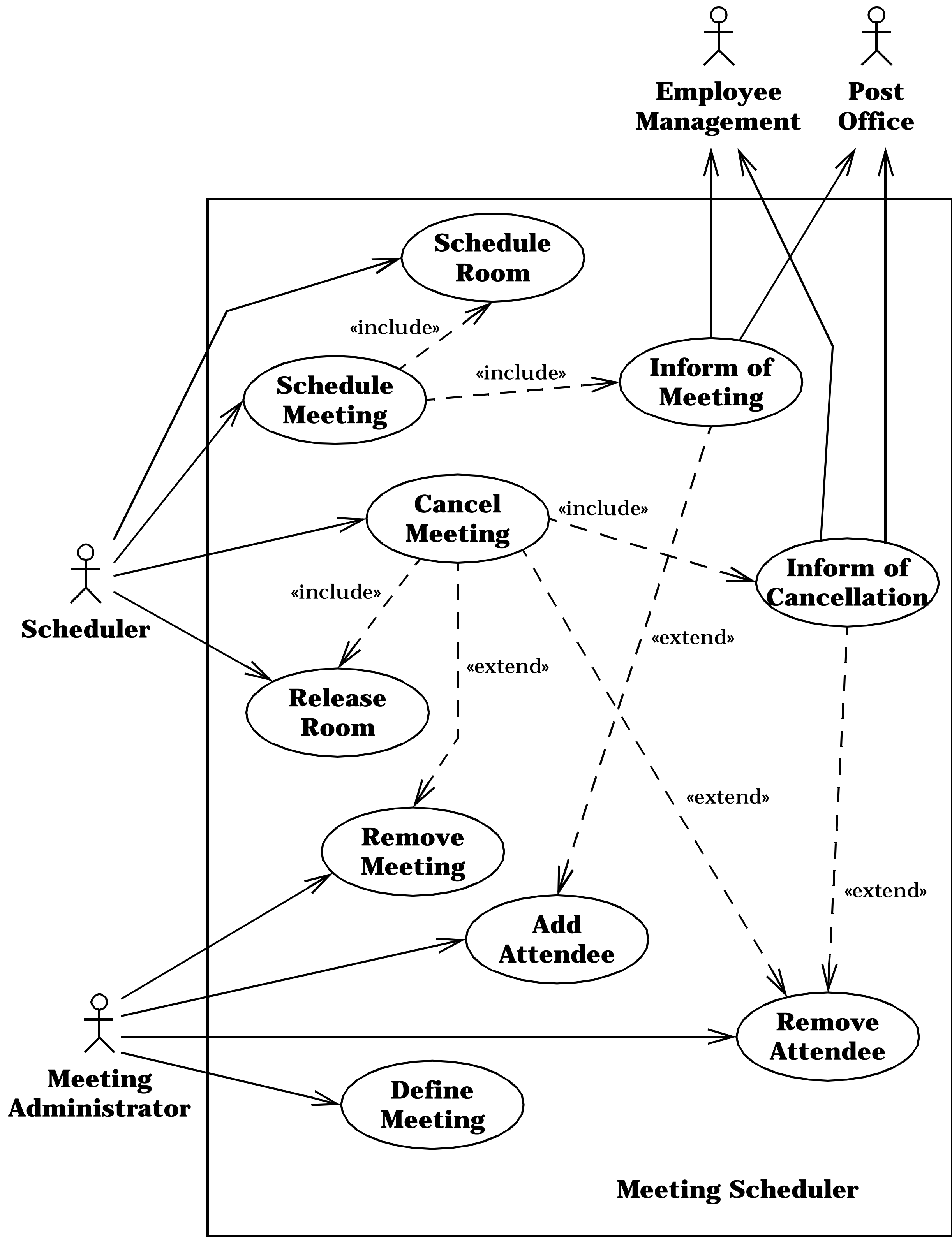

Figure 2b: A more detailed (UML 1.3) use case diagram.

© 1999-2007 by Objective Engineering, Inc.

Postconditions:

- There exists a definition of this meeting, m, in the system.
- m's attendee list includes the attendees provided to the use case.

The Add Attendee use case is employed to add an attendee to an existing meeting definition. The definition of the use case is:

Trigger:

- an event, `addAttendee(meetingName, attendee: employeeId)`

Preconditions:

- There exists m, a definition for a meeting named name, in the system.

Postconditions:

- m's attendee list includes the attendee provided to the use case.
- If any future scheduled instances of `m` exist, then apply the extension Inform of Meeting to inform the attendee of each such instance.

The Add Attendee use case is extended by the Inform of Meeting use case. That latter use case's specification is:

Trigger:

- an internal condition,
  its use by Schedule Meeting or extension of Add Attendee
  (to inform an employee indicated by `employeeId` about a meeting `m` during time period `p`)

Preconditions:

- none

Postconditions:

- an invitation to meeting `m` during period `p` is sent to the electronic mail address of the indicated employee

The Remove Attendee use case removes an attendee from an existing meeting definition. Its specification is:

Trigger:

- an event,
  `removeAttendee(meetingName, attendee: employeeId)`

Preconditions:

- There exists m, a definition for a meeting named `name`, in the system.
- The attendee provided to the use case is an attendee of that meeting.

Postconditions:

- m's attendee list does not include the attendee provided to the use case.
- If any future scheduled instances of `m` exist, then apply the extension Inform of Cancellation to inform the attendee of each such instance.
- If, after removing the attendee, m's attendee list is empty, and if future scheduled instances of m exist, the apply the extension Cancel Meeting to cancel those instances.

The Remove Meeting use case removes a meeting definition from the system. Furthermore, if any scheduled instances of that meeting exist, they must be canceled. (The Cancel Meeting use case is therefore an extension of this use case.) The Remove Meeting specification is:

Trigger:

- an event, `removeMeeting(meetingName)`

Preconditions:

- There exists m, a definition for a meeting named `name`, in the system.

Postconditions:

- m no longer exists in the system.
- For any scheduled instance, d, of meeting m:
  `cancelMeeting(d, st)`,
  where st is the scheduled starting time of d.

The Inform of Cancellation use case extends the Remove Attendee use case. Its specification is:

Trigger:

- an internal condition,
  its use by Cancel Meeting or extension of Remove Attendee
  (to inform an employee indicated by `employeeId` about a meeting `m` during time period `p`)

Preconditions:

- none

Postconditions:

- a retraction of the invitation to meeting `m` during period `p` is sent to the electronic mail address of the indicated employee

The Schedule Room use case schedules a room of a specified size for a specified time period. Its specification is:

Trigger:

- an event, `scheduleRoom(size, timePeriod): roomNumber`

Preconditions:

- There exists a room, r, such that r's size is at least as large as `size`, and r is available during `timePeriod`.

Postconditions:

- For the *smallest* room, r, such that r's size is at least as large as `size` and r is available during `timePeriod`:
  r is no longer available during `timePeriod`, and
  return r's room number

The Release Room use case releases a specified room during a specified time period. The definition of Release Room is:

Trigger:

- an event, `releaseRoom(roomNumber, timePeriod)`

Preconditions:

- The room, r, whose room number is `roomNumber` is not available during `timePeriod`.

Postconditions:

- The room, r, whose room number is `roomNumber` is available during `timePeriod`.

To schedule a meeting, you employ the Schedule Meeting use case. That use case schedules a specified meeting for a specified time period. Its specification is:

Trigger:

- an event, `scheduleMeeting(name, timePeriod)`

Preconditions:

- There exists a defined meeting, m, in the system such that m's name is `name`.
- There is no instance of m scheduled during `timePeriod`.

Postconditions:

- There is a scheduled instance of m during `timePeriod`.
- m is assigned a room determined by `scheduleRoom(size, timePeriod)`, where size is the number of attendees of m.
- Each attendee of m is informed of the scheduled meeting through the Inform of Meeting use case.

The Cancel Meeting use case cancels a specified meeting with a specified starting time. Its specification is:

Trigger:

- an event, `cancelMeeting(name, startTime)`

Preconditions:

- There exists a defined meeting, m, in the system such that m's name is `name`.
- There is a scheduled instance of m that will start at `startTime`.

Postconditions:

- The scheduled instance, i, of m starting at `startTime` has been removed from the system.
- i's room is released using `releaseRoom(roomNumer, timePeriod)`, where `roomNumber` is the room assigned to i, and `timePeriod` is i's time period.
- Each attendee of m is informed of the canceled meeting through the Inform of Cancellation use case.

© 1999-2007 by Objective Engineering, Inc.

These specifications indicate exactly what each use case must accomplish and therefore can be a valuable resource when developing the design.

The Design

You are provided a Post Office package with Post Office and Address classes. You can also assume the existence of an Employee Management component with an Employee Management facade class and an Employee interface class. Figure 3 depicts those entities.

What classes are required by the scheduling software? If you enumerate the tangible things, roles, events, and interactions in the problem domain, you arrive at the following list:

Meeting. An instance of this class represents a meeting that can be scheduled again and again. Its attributes include a name and a list of the attendee's employee identifiers.

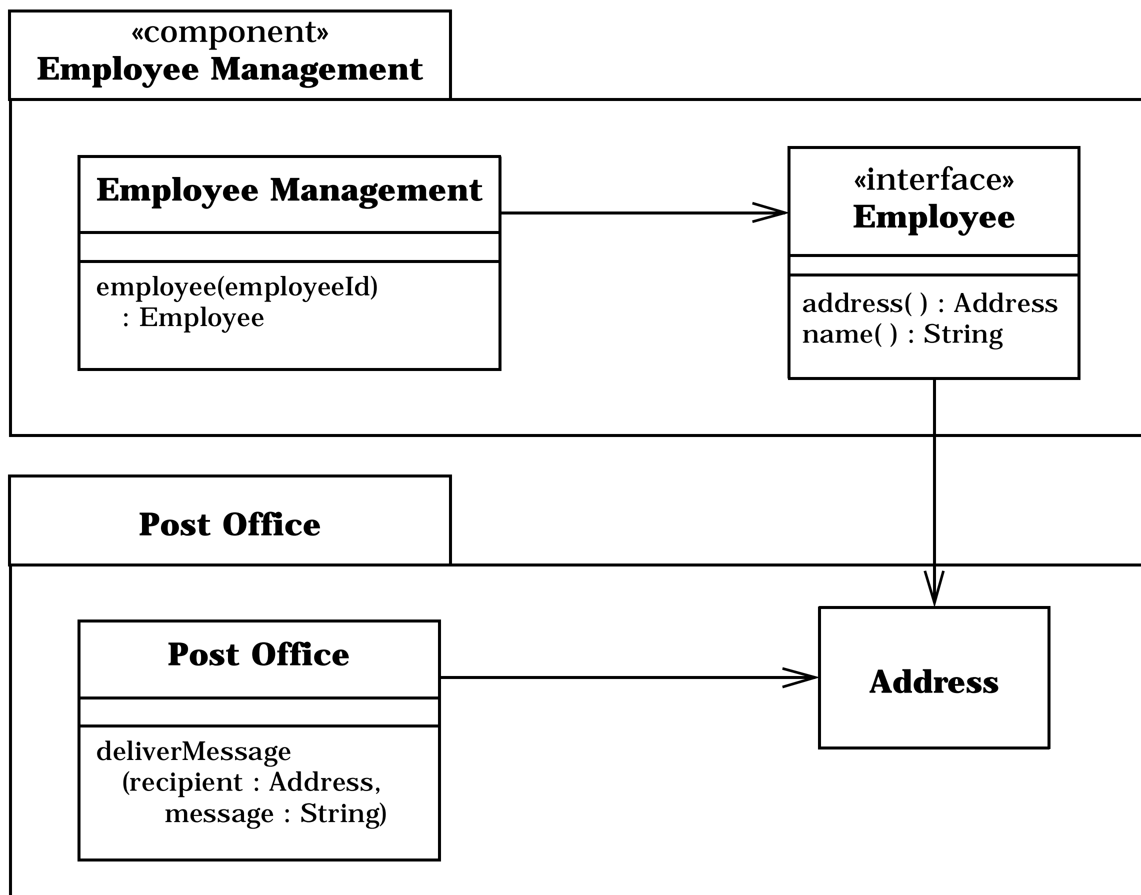

Figure 3: Existing packages.

© 1999-2007 by Objective Engineering, Inc.

Scheduled Meeting. A Scheduled Meeting is a scheduled instance of a Meeting. It has a starting and ending time.

Room. A Room instance models a meeting room. It has a capacity (the number of people the room will hold) and a location (most likely a room number).

Room Assignment. An instance of this class represents the scheduling of a Room for a particular temporal interval. It includes a starting and ending time.

Figure 4 depicts these classes and their attributes.

The classes in Figure 4 provide a starting point. By what process can you elaborate this design? Given the existence of use case specifications, an obvious approach is to employ those descriptions to guide the static design. In particular, you analyze each specification, adding the properties to the class diagram required to check the preconditions and effect the postconditions of the use case.

Consider the first use case, Define Meeting. This use case is triggered when a user playing the role of Meeting Administrator issues a request to define a meeting. How shall that user access the scheduling system to make that request? Let's assume that all client (GUI) interfaces access meeting scheduling application objects by directly invoking methods on those (perhaps remote) objects. (A later portion of this document revisits this assumption.) The most obvious solution is the use of a *facade* that serves as an API class [GHJ&V, pp. pp.185-193].

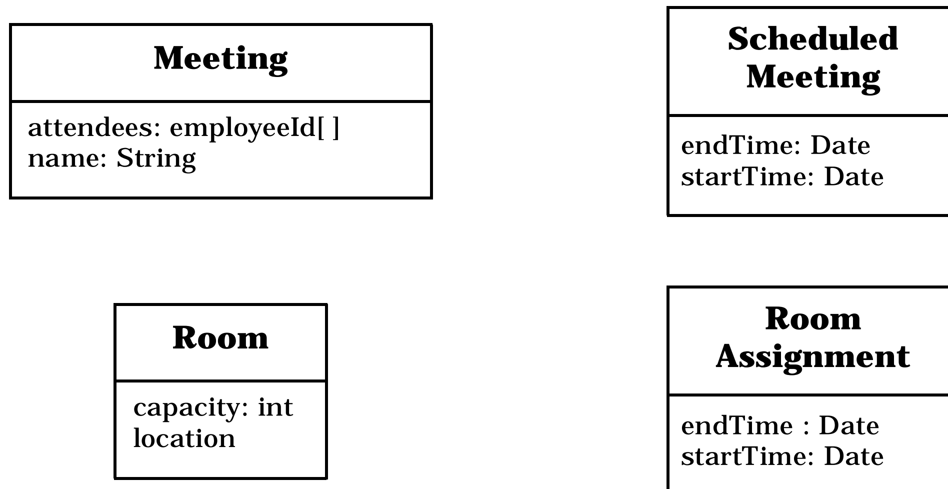

Figure 4: An initial group of classes.

© 1999-2007 by Objective Engineering, Inc.

Figure 5 depicts that facade class. For now, assume that this class is used only to create, alter, and remove meeting definitions. The class defines a method for each of the meeting definition use cases (Define Meeting, Add Attendee, Remove Attendee, Remove Meeting). Although Figure 5 omits the full signatures of those methods, the parameters of each are outlined in the use case specifications in the requirements model. The `defineMeeting` method, for example, takes as arguments the name of the meeting and a list of employee identifiers that indicate the attendees of the meeting.

> **Note:** As a general object-oriented design guideline, you should avoid the use of "god classes" that act as centralized controllers [Riel, pp.32-36]. Furthermore, each class should be cohesive in that it has just one general responsibility. At first blush, a facade class, such as the Meeting Administration class, may appear to violate both of these guidelines, and indeed you may suffer a temptation to dump controlling behavior in a facade. A properly conceived facade, however, merely forwards requests to the proper underlying instances, and is therefore neither incohesive nor a god object.

The precondition of the Define Meeting use case is that a meeting of the specified name must not already exist. This implies that some object must be able to search for a Meeting using a meeting name as a key. The Meeting Administration is the obvious instance in which to place this responsibility, resulting in the qualified association shown in Figure 6.

Once the Meeting Administration instance verifies that no Meeting with the specified name already exists, it creates the new Meeting instance. (You might delegate the responsibility to create Meetings to another entity, such as a Meeting Factory, but that entity is not included at this level of the design.) Because a Meeting maintains its attendee list as a set of employee identifiers, that physical implementation is reflected in the design. In a logical design, the Meeting class has a one-to-many association with the Employee class, and the

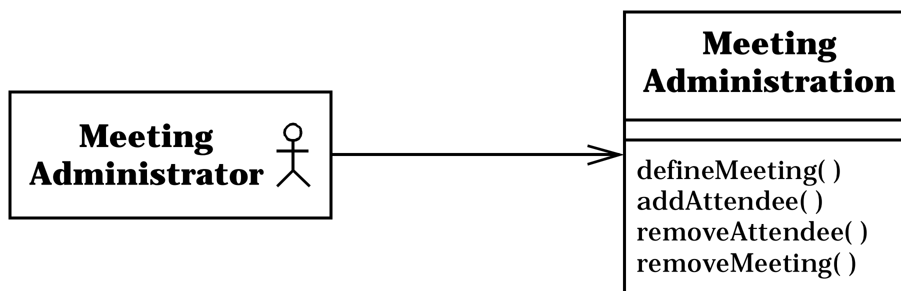

Figure 5: The Meeting Administration facade class.

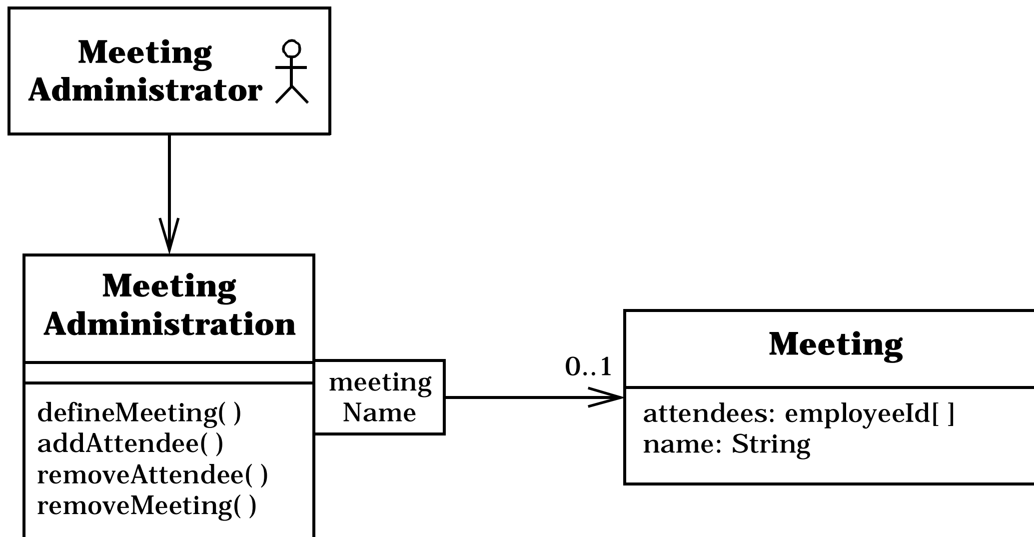

Figure 6: Locating a Meeting using a meeting name.

employee identifier is an attribute of that latter class. Because the Employee interface class is a part of the Employee Management component, however, and because that component locates Employees using employee identifiers as keys, this design does not employ that logical representation.

The postconditions of the Define Meeting use case are that a Meeting instance must be created, and that its attendees must include the set of attendees specified by the user. Both are achieved by creating (and retaining) an instance of the Meeting class that appears in Figure 6. Figure 7 depicts a scenario for defining a meeting. The Meeting Administration instance must attempt to find an existing Meeting instance with the specified name, although the mechanism employed to conduct that search is not included in the figure.

The Add Attendee and Remove Attendee use cases alter the attendee list of an existing Meeting. Recall that Inform of Meeting extends Add Attendee, meaning that when an employee is added as an attendee of a Meeting, he or she must be informed of any future Scheduled Meetings for that Meeting. Figure 8 contains a sequence diagrams for the simplest scenario for adding an attendee — the case where no future Scheduled Meetings exist for this Meeting. The sequence diagram requires that an `addAttendee` method be introduced in the Meeting class. (The Meeting Administration facade class already has `addAttendee` and `removeAttendee` methods, and it has an association with the Meeting class. These features are also required by the sequence diagrams for adding and removing attendees.)

© 1999-2007 by Objective Engineering, Inc.

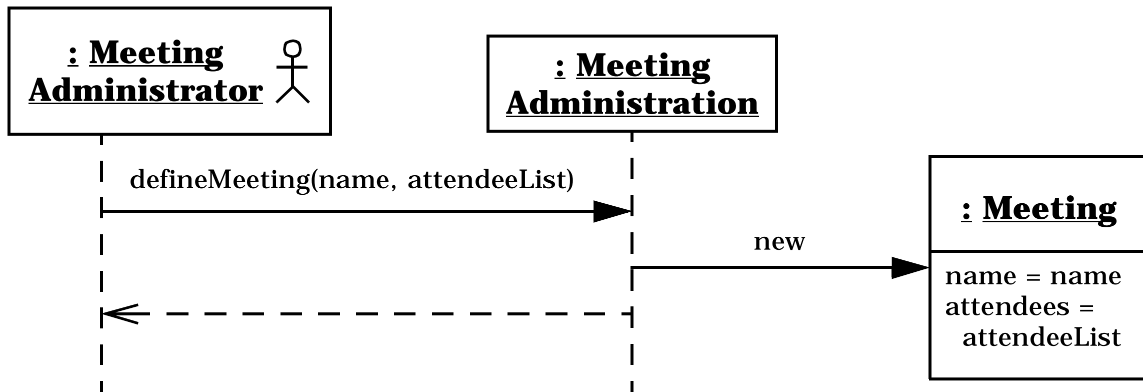

Figure 7: A meeting definition scenario.

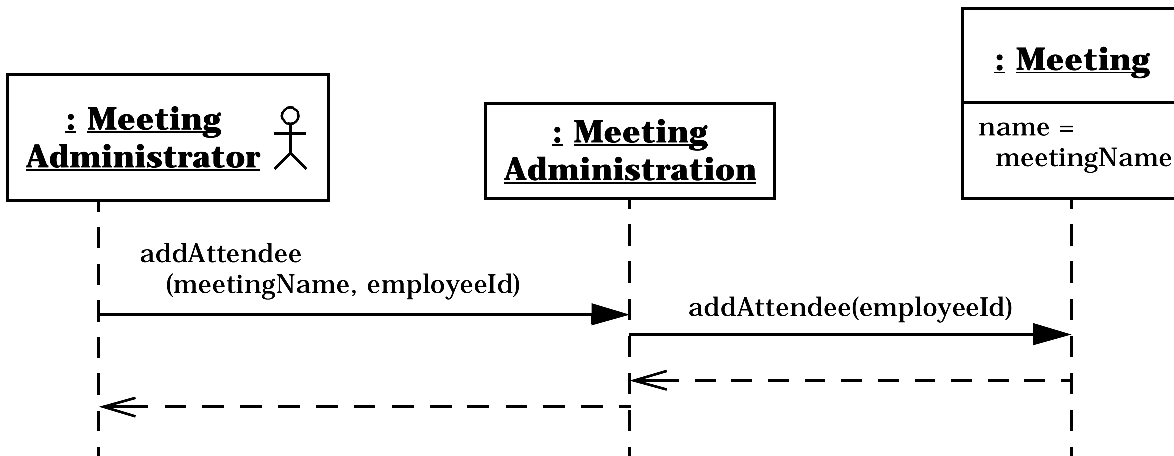

Figure 8: The simplest scenario for adding an attendee.

The Inform of Meeting use case extension occurs if the Meeting has future Scheduled Meetings. To determine whether this condition holds, each Meeting instance must maintain a list of those Scheduled Meetings. Furthermore, either the Meeting or its Scheduled Meetings must inform the attendee. If a Meeting assumes that responsibility, then it must query its Scheduled Meetings for their periods (i.e., their starting and ending times). If a Scheduled Meeting performs that task, then it must obtain the meeting name (by retaining that information, by querying its Meeting to obtain the name, or by receiving the meeting name as a parameter to its `addAttendee` method).

Neither option is obviously superior to the other. In this design, the responsibility to inform attendees is assigned to Scheduled Meeting instances. Figure 9 includes the required additions to the class diagram. The meeting name and attendee list are retained in each Scheduled Meeting as well as in the Meeting. (The latter is not required here, but will be required when canceling a Scheduled Meeting.) The `addAttendee` method is added to both classes, and a Meeting retains an ordered list of its Scheduled Meetings. A Scheduled Meeting must use the Post Office and Employee Management packages to obtain addresses and send electronic mail.

Recall from Figure 3 that the Employee Management and Post Office packages have facade classes of the same name that provide access to the package. How does a client, such as a Scheduled Meeting instance, obtain a reference to those facade objects? Although not depicted in Figure 3, the Employee Management and Post Office classes are implemented using the Singleton pattern [GHJ&V,

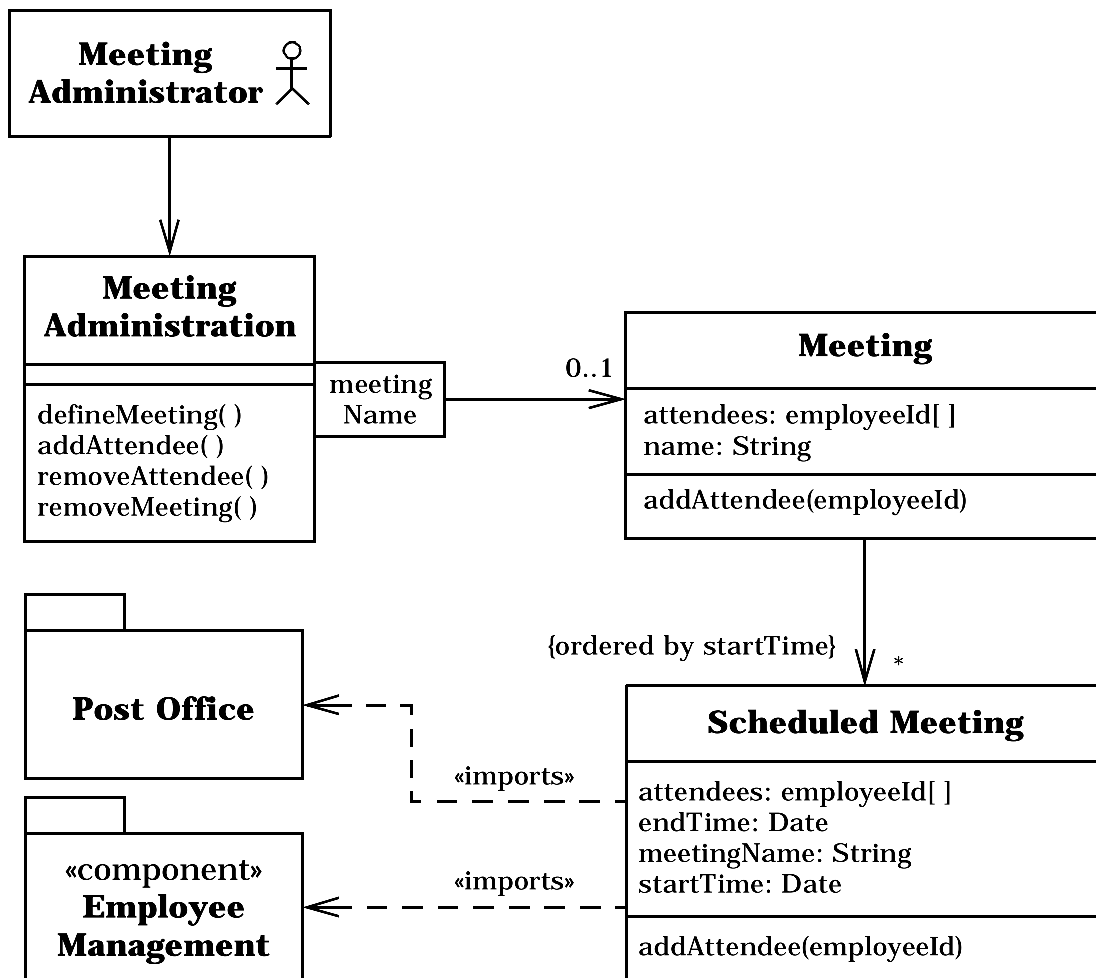

Figure 9: Extending the class diagram for the Inform of Meeting use case.

© 1999-2007 by Objective Engineering, Inc.

pp. 127-134]. To obtain access to the single instance of either class, a client calls the `instance` class method, which returns a reference to that single instance.

The collaboration diagram in Figure 10 illustrates a scenario in which an attendee is added to a Meeting with one Scheduled Meeting. When a class method is invoked, the class itself can be included in a collaboration or sequence diagram. As a result, the figure includes the Post Office and Employee Management *classes* as well as the single instance of each. Recall also that PostOffice::PostOffice refers to the Post Office class in the Post Office package, and so the Post Office class and its instance in the diagram are obtained from that package. The figure uses the package notation to depict explicitly that the Employee Management and Employee types are defined in the Employee Management package.

The behavior of the Remove Attendee use case is analogous to that of Add Attendee. Although not included here, the interaction diagrams for removing an attendee from a Meeting with no future Scheduled Meetings, and from a Meeting with one future Scheduled Meeting, are very similar to the diagrams in Figures 8 and 10. The major difference is that a `removeAttendee` method (rather than the `addAttendee` method) must be invoked in the Meeting and Scheduled Meeting instances. Figure 11 contains a class diagram with those additions.

If the last attendee is removed from a Meeting, the Meeting must cancel its Scheduled Meeting instances for which the start times are in the future. This implies that the Scheduled Meeting class must include a `cancel` instance method that releases the Room assigned to that instance. Figure 11 includes the `cancel` method. The details of how it releases a Room are deferred until the discussion of the Cancel Meeting use case.

The Remove Meeting use case removes a meeting definition from the system. If that meeting is scheduled to be held anytime in the future, however, those future meetings must be canceled. It might be best to postpone consideration of this use case until the Schedule Meeting use case is analyzed. The Schedule Meeting use case in turn uses the Schedule Room use case, so perhaps the most prudent course at this point is to design the scheduling of a room.

Unlike the use cases already considered, the Schedule Room use case is invoked by an instance of the Scheduler actor. (This actor represents a person playing the role of a meeting or room scheduler — that is, someone requesting the scheduling of a meeting or room.) The Scheduler and Room Administrator actors will be implemented as distinct graphical user interfaces. To increase cohesion and reduce coupling, therefore, a separate facade class is presented to each.

© 1999-2007 by Objective Engineering, Inc.

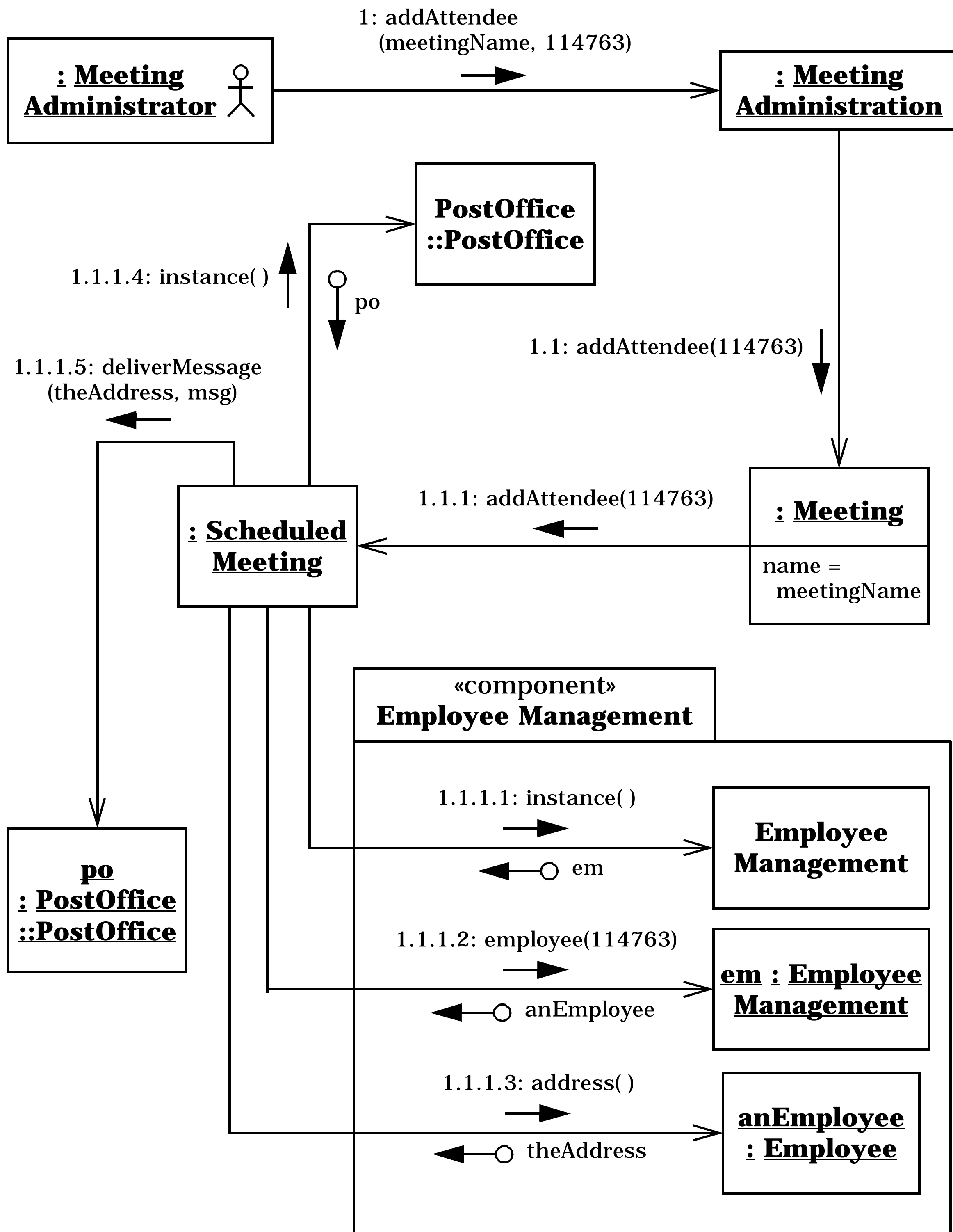

Figure 10: A more complicated scenario for adding an attendee.

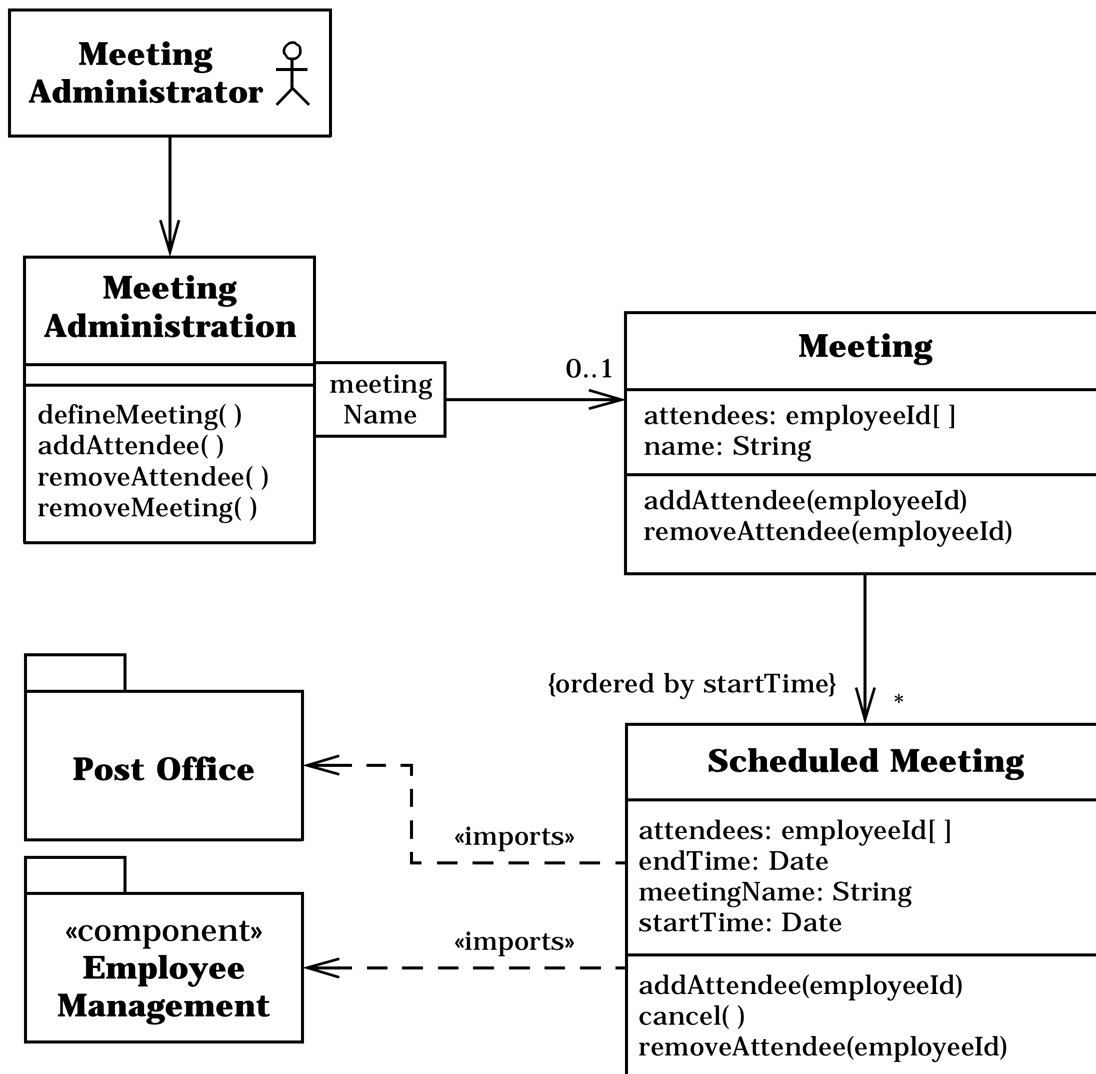

Figure 11: Extending the class diagram to handle removing attendees.

The Facilities Scheduler facade is used by a Scheduler actor to schedule meetings and rooms, as well as to cancel meetings and room assignments. It therefore requires methods for those four tasks. Figure 12 depicts the facade class. Although they not specified in the figure, the signatures of those methods are as stated in their corresponding use case specifications. For example, the `scheduleMeeting` method takes a room size and a time period as arguments and returns a room number. (It may also throw an exception if no room of that size is available during that period.)

© 1999-2007 by Objective Engineering, Inc.

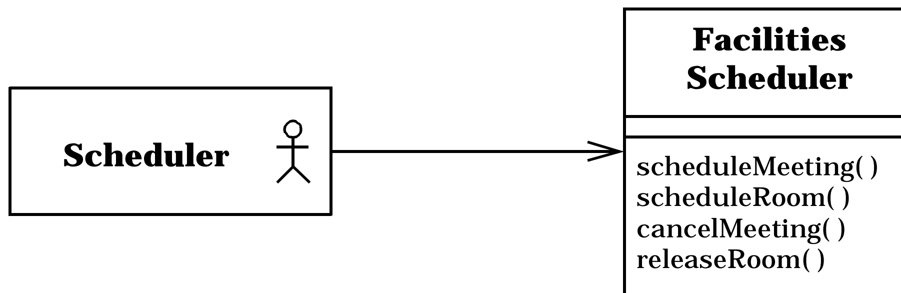

Figure 12: The Facilities Scheduler facade class.

To handle the management and scheduling of Rooms, this design includes a Room Pool class. The Room Pool class is a singleton — a single instance of it will exist during system execution. (The Facilities Scheduler facade may also be a singleton; alternatively, each Scheduler actor might be linked to a distinct facade instance.) The Room Pool maintains a list of Room instances organized by room size. It provides methods to schedule a Room of a given size for a given period, and to release a Room of a specified room number during a specified period.

Recall that a postcondition of the Schedule Room use case states that the *smallest* available room of sufficient size must be assigned. To schedule a Room, the Room Pool iterates over all the Rooms that are at least as large as the requested size, starting with the smallest and proceeding in size order. It asks each Room to schedule itself during the specified period. If a Room indicates that it has scheduled itself, the Room Pool returns that room number.

To schedule itself, a Room must maintain a list of its Room Assignments. As it schedules itself, it adds a new Assignment. (How and when Assignments are removed is ignored here.) The class diagram in Figure 13 illustrates the various features required for this use case.

Figure 14 contains a collaboration diagram in which a room is scheduled successfully. The diagram omits the details of how a Room checks its Room Assignments. In this scenario, the second Room checked by the Room Pool is available. The Pool then obtains the Room's location (room number 604) and returns that to the facade, which in turn returns it to the actor instance.

When no Room is available, the Room Pool must raise an error condition. In languages that provide support for exceptions (such as C++ and Java), the Room Pool's `scheduleRoom` method could throw an exception that (most likely) will be passed through the Facilities Scheduler to the actor. The Scheduler actor will catch the exception and display an appropriate message. (Recall that

© 1999-2007 by Objective Engineering, Inc.

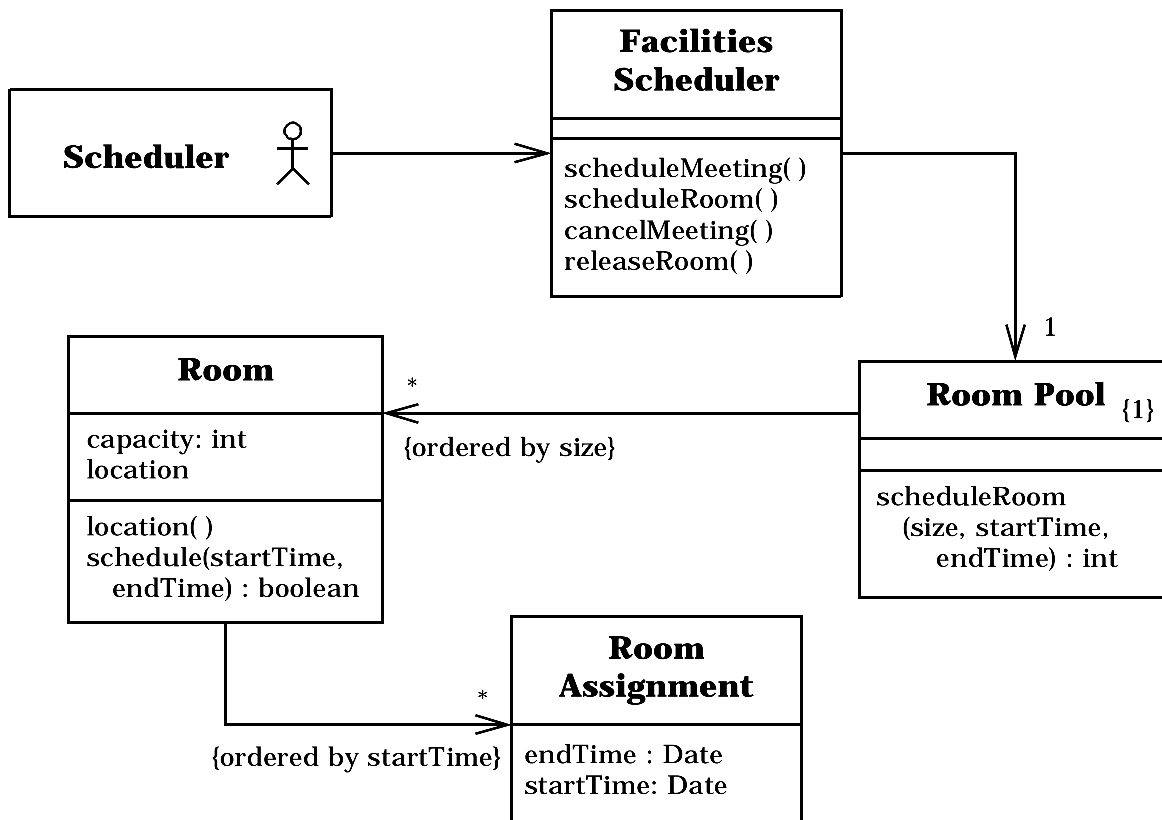

Figure 13: The class diagram for scheduling a room.

the two actors will be implemented as presentation interfaces, such as graphical user interfaces.) In languages without an exception mechanism, a simple hack is to return a negative (or otherwise illegal) room number. A more elegant approach is to return an object that contains an indication of success (or failure) and, in the case of success, the assigned room number.

The Schedule Meeting use case, given a meeting name and a time period, must attempt to schedule the specified meeting for the specified time. The `scheduleMeeting` method in the Facilities Scheduler facade must first obtain access to the Meeting instance with the specified name. Recall that the Meeting Administration facade object maintains a list of Meetings qualified by meeting name. One possible way to provide this access, therefore, is through the definition of a `meeting` method in that facade that, given a meeting name, returns a reference (or pointer) to a Meeting instance. This is a simple solution, although it has one potentially negative side: that interface is available to all clients of the Meeting Administration facade, including the Meeting Administrator actor class. This disadvantage seems rather harmless, however, and so that solution is adopted here.

© 1999-2007 by Objective Engineering, Inc.

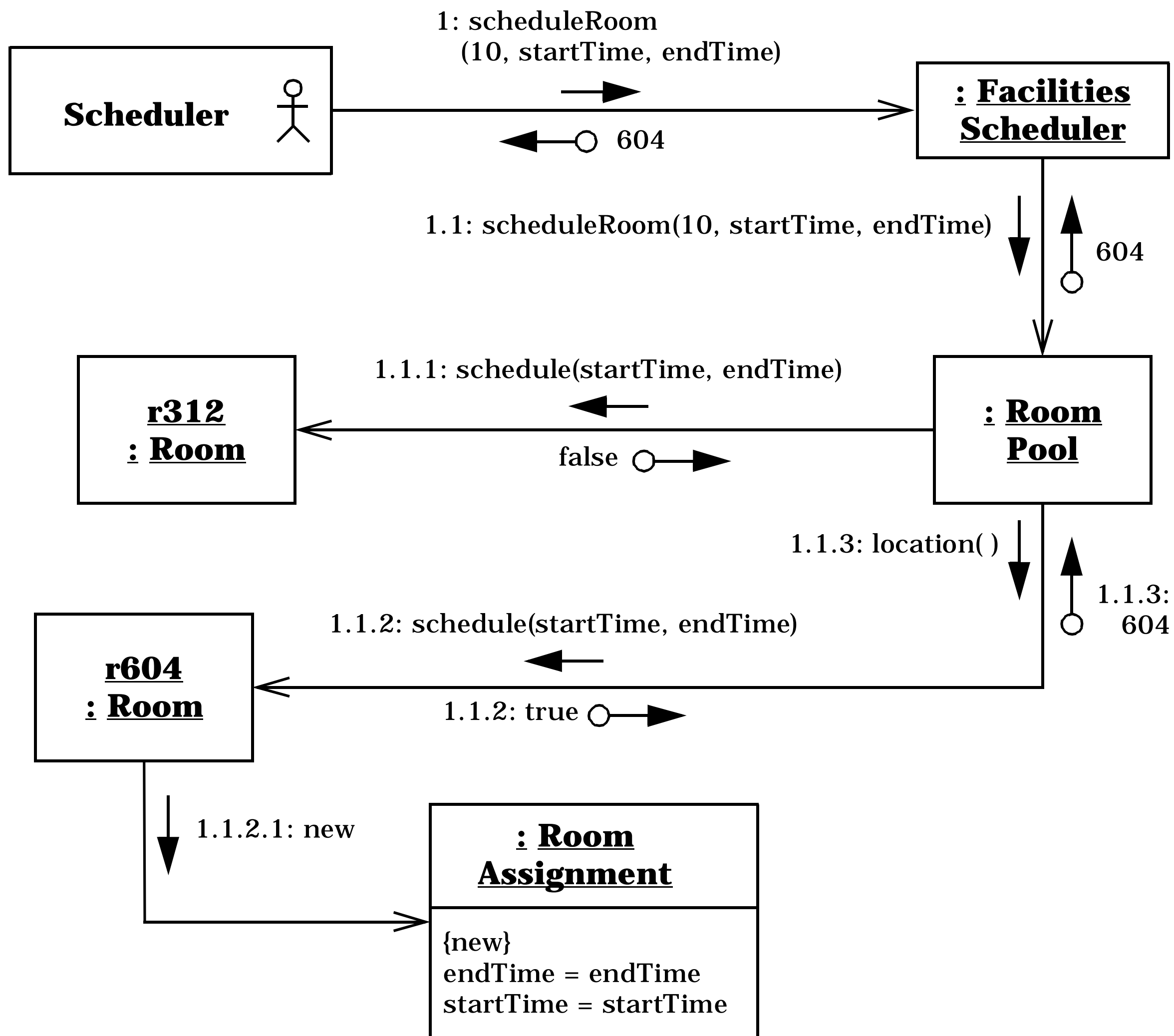

Figure 14: A scenario for scheduling a room.

(An alternative solution is to provide two facades for meeting administration. One, the existing Meeting Administration facade, is available externally and provides the methods required by the Meeting Administrator actor. The other, available to the Facilities Scheduling facade, provides the method interfaces required by that facade.)

After obtaining a Meeting reference, the Facilities Administration instance asks the Meeting to schedule itself for the specified time period. The Meeting must use the Room Pool to (attempt to) schedule a Room, after which it must create a Scheduled Meeting instance. As the final step, that Scheduled Meeting instance must inform the attendees of the meeting.

To achieve the latter behavior, the general approach introduced in Figures 9 and 10 can be used — a Scheduled Meeting informs attendees about the meeting. This implies that when it is created, a Scheduled Meeting is given a list of attendee employee identifiers; it iterates through that list, informing each attendee of the meeting and its location. This in turn implies that a Scheduled Meeting must be linked to the Room to which it is assigned. (Such a link is also required so that a Scheduled Meeting can release its Room when is canceled.)

Because a Scheduled Meeting must be linked to its Room, and because the Room Pool returns the room *number* of the assigned Room, the Pool must provide a method (call it `room`) that, given a room number, returns the Room with that number. The class diagram in Figure 15 contains the various features required to handle this use case.

The collaboration diagram in Figure 16 illustrates the object interactions for a simple scheduling scenario. The diagram ignores the details of how the Room Pool schedules the Room, which is the behavior of the Schedule Room use case. A scenario of that use case is depicted by the collaboration diagram in Figure 14. Its presence in the Schedule Meeting use case is due to the fact that the Schedule Meeting use case `«uses»` (or `«includes»`) the Schedule Room use case.)

In the Release Room use case, a Scheduler actor instance invokes the Facilities Scheduler facade's `releaseRoom` method, specifying a room number and a time period. The facade object forwards that request to the Room Pool. The Room Pool then looks up the Room with the specified room number and asks that Room to release itself for the specified period. The class diagram Figure 17 contains the additional methods required to handle this use case.

The steps for canceling a meeting are somewhat similar to those for scheduling a meeting. The Meeting must be located, after which it is asked to cancel itself during the specified period. The Meeting class therefore requires a `cancel` instance method. The Meeting object locates the appropriate Scheduled Meeting instance and invokes its `cancel` method (thus requiring a `cancel` method in that class). The Scheduled Meeting instance in turn invokes the `release` method in its Room.

Figure 18 contains the final class diagram. The only properties absent from that diagram are a Scheduled Meeting's association with the external Post Office and Employee Management packages (or, to be more specific, with the Post Office and Employee Management facade classes in those packages). The dynamic model for the system includes the interaction diagrams described in this document, as well as others that have not been included here.

The classes in a system can be grouped into logical units such as subsystems and layers. A group of classes is represented as a package in a UML class diagram. The classes in Figure 18, for example, could be organized into

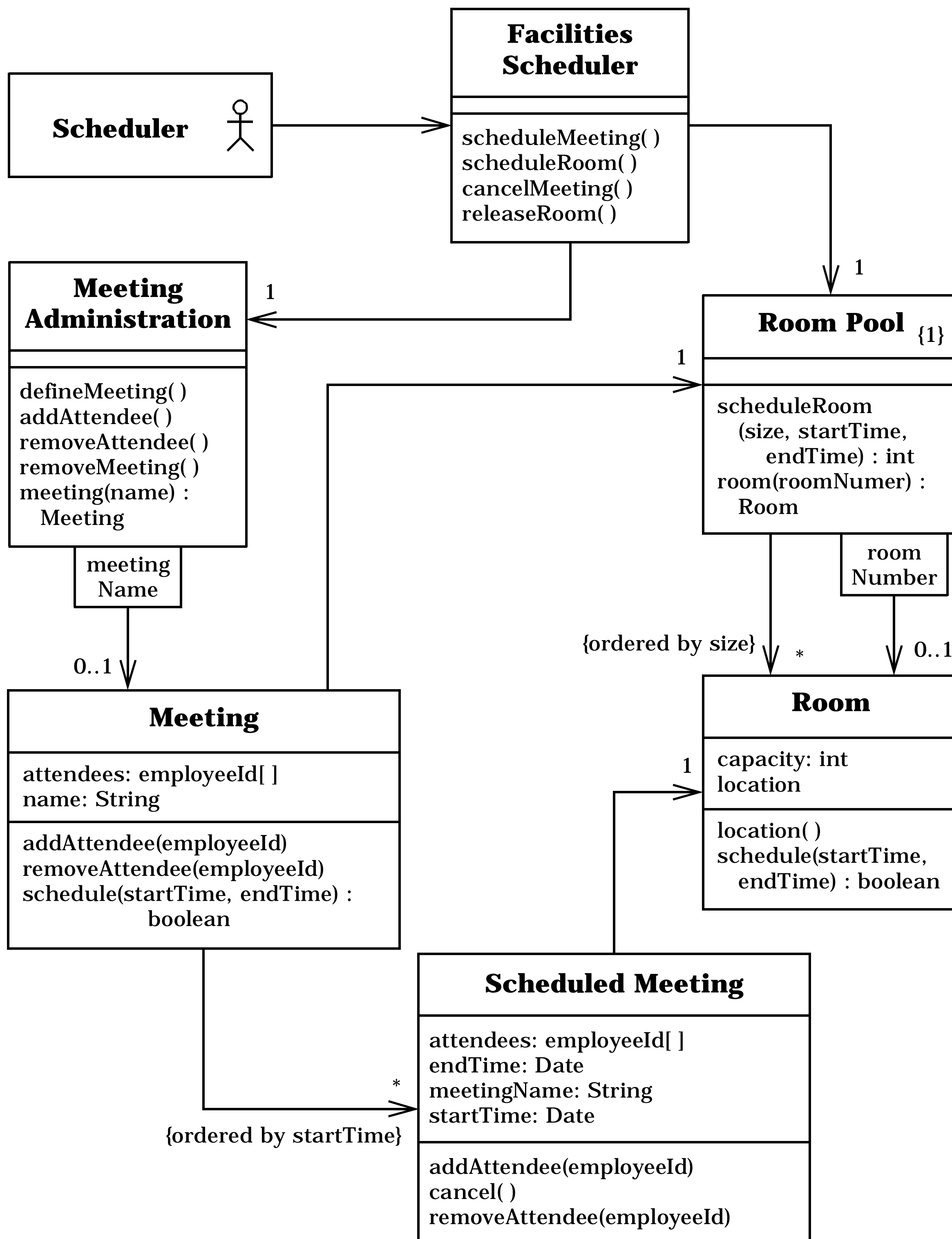

Figure 15: The class diagram for scheduling a meeting.

© 1999-2007 by Objective Engineering, Inc.

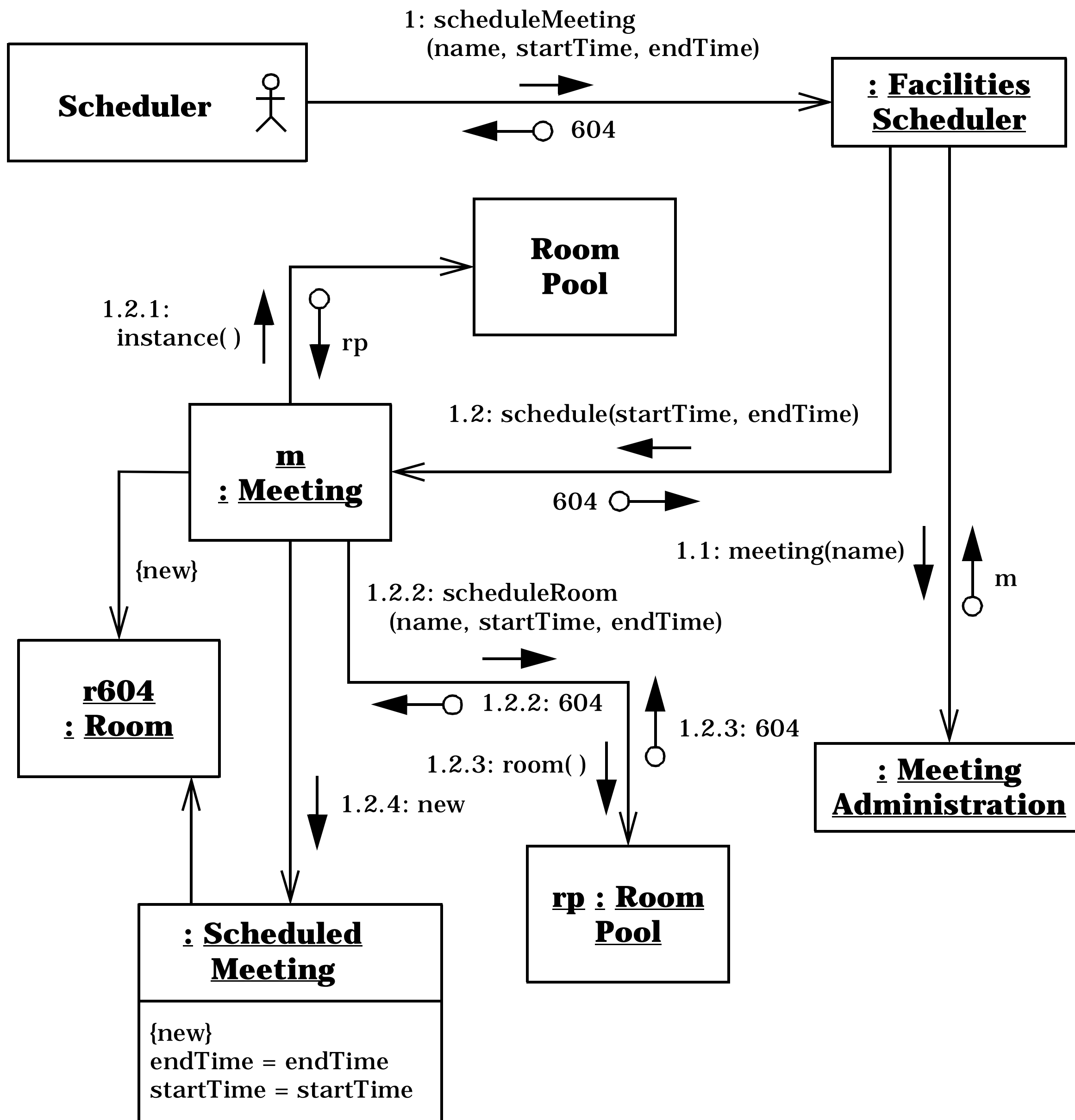

Figure 16: A scenario for scheduling a meeting.

packages that define general functional areas, such as meeting management and room management. The meeting classes would reside in the package for the former, whereas the room classes would belong to the latter.

Security Issues

This solution ignores certain security issues, such as guaranteeing that only the client who scheduled a meeting or room can cancel that assignment. This guarantee could be implemented by associating a token with each assignment

© 1999-2007 by Objective Engineering, Inc.

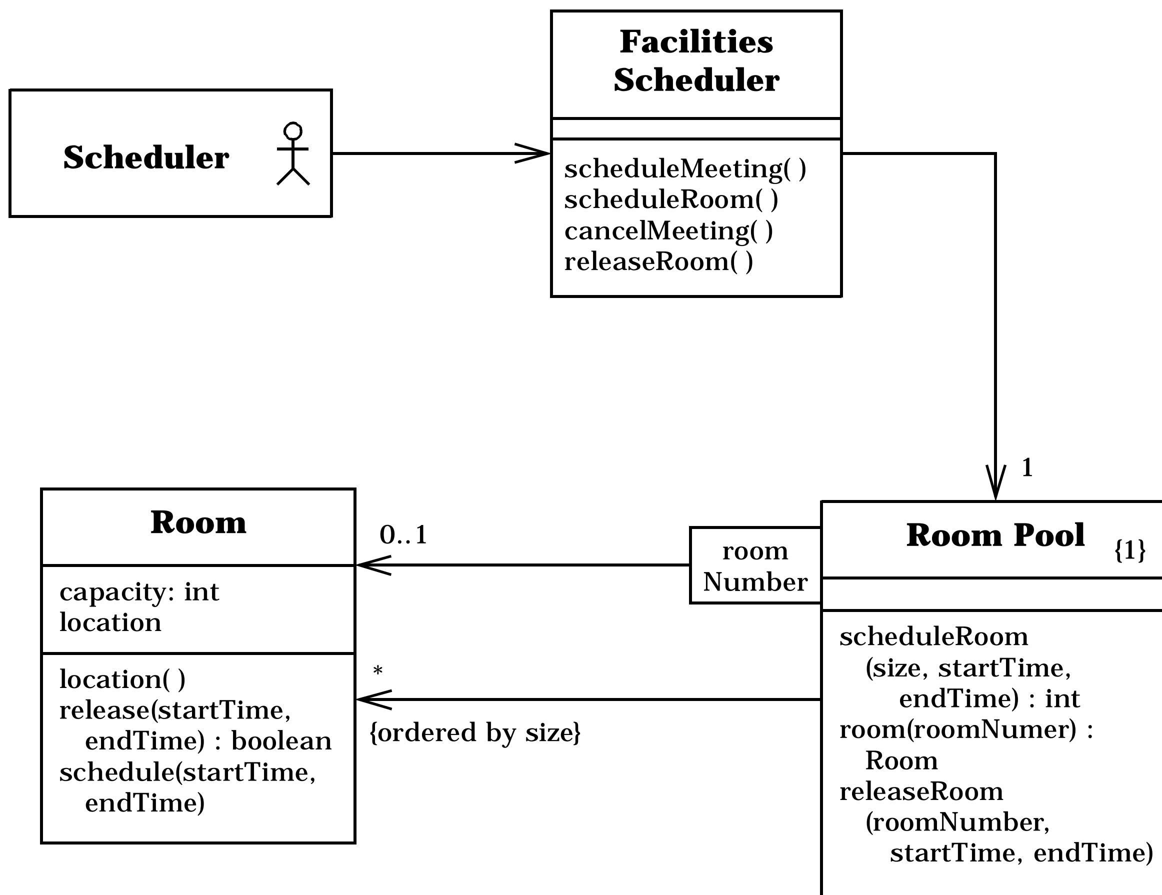

Figure 17: The class diagram for releasing a room.

(and returning that token to the client when the assignment is made), and then requiring that a client present the token when canceling the meeting or room. A disadvantage of this approach is that the client must retain the token. What if an individual who schedules a meeting from one client machine subsequently attempts to cancel that meeting from a different machine? This requires some mechanism (such as a token database) that permits tokens to be accessed from any client machine.

Perhaps a superior approach is to associate with each assignment an indication of who requested the assignment. An obvious candidate is the user name provided by the person when logging onto the system. This allows individuals to work from any client machine and requires only that a person exhibit the same user name each time he or she uses the system.

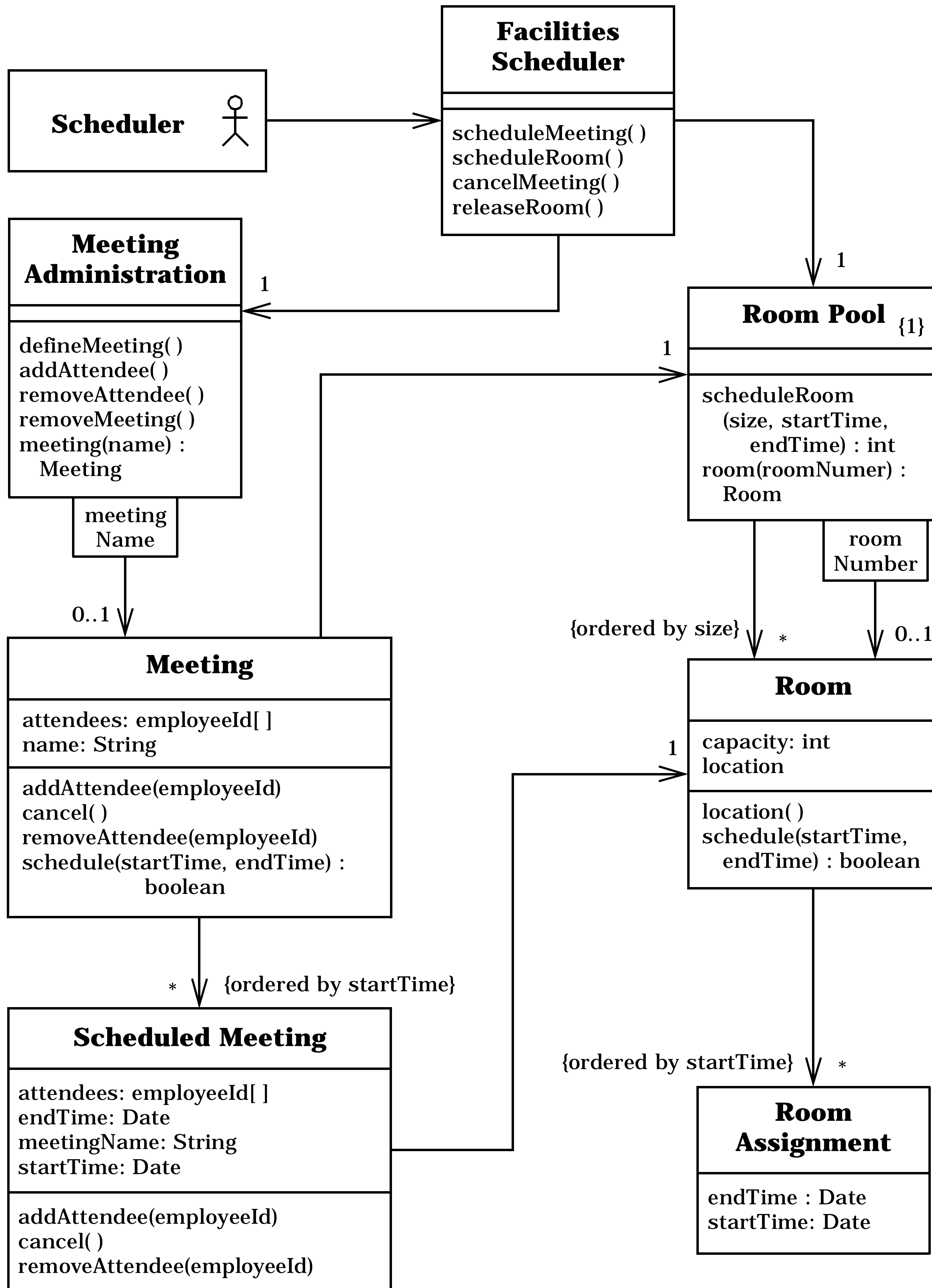

Figure 18: The final class diagram.

© 1999-2007 by Objective Engineering, Inc.

Revisiting the RMI-Based Approach

This design, and particularly its use of facade interfaces, assumes that client (GUI) interfaces use remote method invocations (RMIs) to interact with objects in the meeting scheduling application domain. Suppose, however, that you opt instead for a browser-based interface using HyperText Transfer Protocol (HTTP) requests. This section explores how such a decision would affect the design.

In an HTTP-based solution, a client interface (such as a browser) interacts with an application by issuing an HTTP request. A web server on the application machine receives the request and dispatches it to a server-side entity, such as a Java servlet or a CGI script, responsible for processing such requests. A typical design has one web page for each use case initiated by an actor (as well as a "home page" presented when the actor first initiates the application). Each web page in turn might be associated with one servlet or CGI script that handles requests for that page.

For browser-based clients of the meeting scheduling application, you replace the (RMI-oriented) facades with a servlet-based or CGI-based approach. Figure 19 depicts a portion of a class diagram for the former. The two dependencies in the diagram indicate that each actor will interact with the application through HTTP requests. A web server (not depicted in the diagram) receives each request and forwards it to the appropriate servlet. The "stacking" of the two servlet classes, as well as the generic names of those two classes, are meant to indicate that several scheduling servlets and administration servlets, the specifics of which have yet to be specified, will be a part of this design.

What specific servlets will you have? In general, each use case initiated by an actor maps to a servlet. Hence, the Scheduler and Meeting Administrator actors interact with servlets that correspond to the use cases initiated by those actors in Figures 2a and 2b. (Those servlets may actually be implemented as

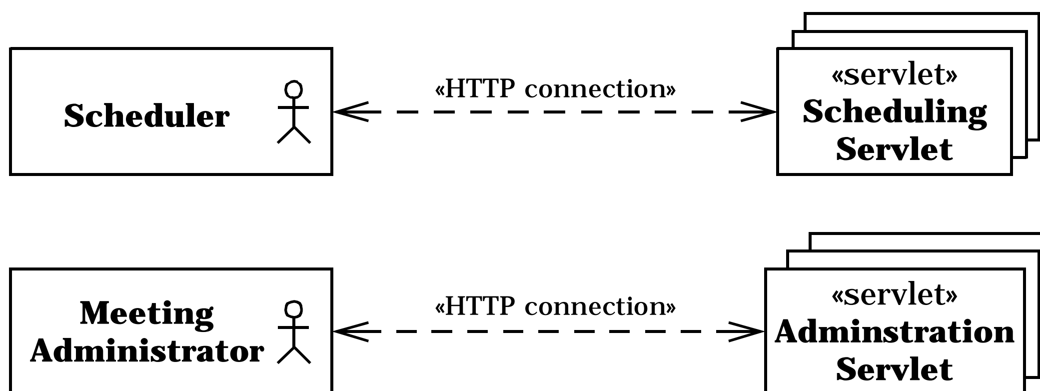

Figure 19: A part of a class diagram for HTTP-based interactions.

© 1999-2007 by Objective Engineering, Inc.

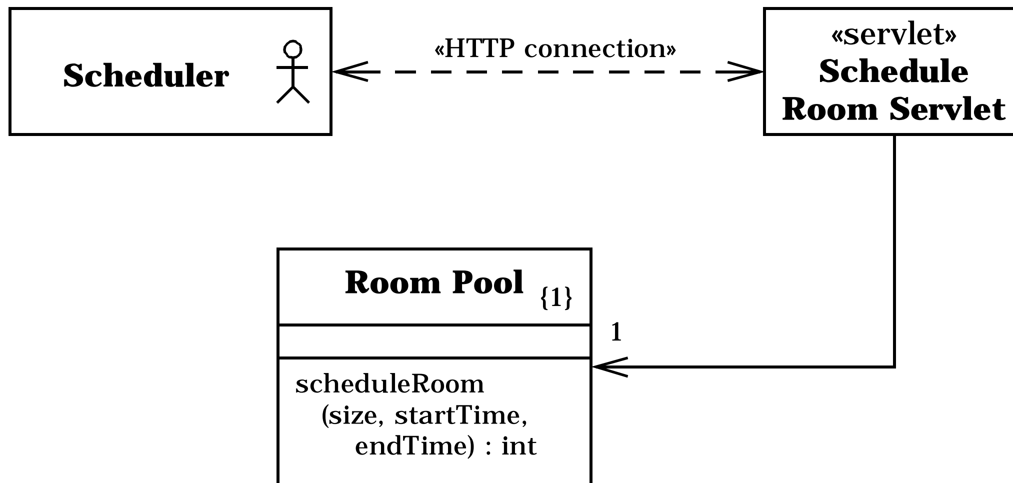

Figure 20: A part of a class diagram for one scheduling servlet.

JavaServer Pages. Additionally, you might use one or more JSPs to present the results of each servlet request. A discussion of the detailed design of servlets and JSPs for this application is beyond the scope of this paper.)

Each servlet might in turn interact with objects in the application domain. Those dependencies appear as normal associations in a UML class diagram. Figure 20 depicts the specific servlet for the Schedule Room use case. Like the Facilities Schedule facade in the previous design, the servlet for this use case interacts with the Room Pool instance to schedule a room. The structure of the Room Pool class is identical to that of the Room Pool class in the previous design. (Figure 20 omits the Room Pool's relationship with Room instances.)

References

[GHJ&V] E. Gamma, R, Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Riel] A. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

Trademarks

Java and JavaServer are registered trademarks of Sun Microsystems. Inc.