CSC 308 Lecture Notes Weeks 7 and 8
Introduction to Fully Formal Specification

I. Formal specification with preconditions and postconditions.

A. As model object and operation definitions take shape, we are ready to formalize the definitions fully.

B. The formal technique we will use in 308 is based on operation *preconditions* and *postconditions*.

1. A precondition is a predicate (i.e., boolean-valued expression) that is true before an operation executes.
2. A postcondition is a predicate that is true upon completion of an operation.
3. Since pre- and postconditions are predicates, this style of formal specification called *predicative*.

C. The pre- and postconditions are used to specify fully what the system does, including all user-level requirements for the system.

D. This formal specification is part of the overall requirements specification process we're following, with these steps:

1. gather user-level requirements via usage scenarios
2. identify objects and operations
3. formalize operations with pre- and postconditions
4. refine user-level requirements based on formal specs
5. refine formal specs based on user-level refinements
6. iterate steps 1-5 until done

E. The "until done" step involves two levels of validation.

1. First, we must validate that the specified system is complete and consistent from the end user's perspective.
   a. That is, the system meets all end-user needs and does so in a way that is wholly satisfactory to the end user.
   b. This is accomplished by continued consultation with the end user.
2. The second level of validation involves completeness and consistency from a formal perspective.
   a. This can be accomplished in a number of ways.
   b. In the case of mechanized specification languages, such as FMSL, some completeness and consistency checking is done using a computer-based analyzer.
   c. Another valuable validation technique is peer review via formal walkthroughs.
   d. Also, there are techniques for formal specification testing, including the postulation and proof of *putative theorems*.
      i. Such theorems define properties of the system that we expect to be true, and which can be proved true formally with respect to the pre- and postconditions.
      ii. We will discuss putative theorems briefly in 308, but not use them.

II. Formal specification maxims.

A. In developing any formal software specification, it is useful to observe the following two maxims:

1. *Nothing is obvious.*
2. *Never trust the programmer.*

B. The first maxim relates primarily to user-level requirements.

1. It is often easy to think that a requirement is sufficiently obvious that it need not be stated formally.
2. The problem with this thinking is that one person's obvious is not always the same as another's.
3. To ensure that a specification is sufficiently precise, stating the "obvious" is necessary.

C. The second maxim is necessary to avoid nasty surprises in an implementation.
   1. In many cases, we might consider an application to be sufficiently simple that we can trust the programmer to get a user-level requirement right if we forget to specify it.
   2. In general, such trust is a bad idea.
   3. It is better for the specifier to maintain a respectfully and cordially adversarial relationship with the implementor.

III. Overview of FMSL predicate notation.

A. Predicates in FMSL use a variant of formal mathematical logic.

B. Available operations include boolean logic, arithmetic, lists, tuples, unions, and strings.

C. These operations are summarized in Table 1.
   1. The predicate logic operators are used in boolean-valued expressions.

Predicate Logic:

Operator	Description
and	logical and
or	logical or
not	logical not
=>	logical implication
<=>, iff	logical equivalence
if-then-else	conditional choice
forall	universal quantification
exists	existential quantification

Arithmetic:

Operator	Description
+	addition
-	subtraction
/	division
*	multiplication
#	length

Lists:

Operator	Description
[e1,...,en]	construction (elementwise)
[e1 .. en]	construction (inclusive range)
L[n]	selection (nth, from 1)
L[m:n]	selection (mth - nth)
+	concatenation
-	deletion
in	membership
#	length

Tuples:

Operator	Description
{e1,...,en}	construction
.	selection

Unions:

Operator	Description
.	selection
?.	tag interrogation

Strings:

Operator	Description
"xxx"	construction
L[n]	selection (nth)
L[m:n]	selection (mth - nth)
+	concatenation
in	membership
#	length
explode	convert to list
implode	convert from list

Relational:

Operator	Description
=	equal
!=	not equal
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Table 1: FMSL Notation Summary.

a. Logical and, or, and not have the same meaning as their equivalents in a programming language, e.g., "&&", "||", and "!" in C and C++.

b. Logical implication and equivalence have their standard logical meanings, per the following truth tables

p	*q*	*p* => *q*
0	0	1
0	1	1
1	0	0
1	1	1

p	*q*	*p* <=> *q*
0	0	1
0	1	0
1	0	0
1	1	1

c. The conditional choice operator has the truth tables:

p	*x*	*y*	*if p then x else y*
0	x	y	y
1	x	y	x

p	*x*	*if p then x*
0	x	nil
1	x	x

where expressions *x* and *y* must have the same type.

d. The universal and existential quantifiers have their standard logical meanings, but will be applied in specific ways, as upcoming examples illustrate.

2. The arithmetic operators are used in numeric-valued expressions.
   a. Addition, subtraction, division, and multiplication have their standard mathematical meanings.
   b. The length operator returns the number of digits in an integer value.
3. The list operators are used with values of list-type objects, i.e., objects declared with `'*'` composition.
   a. The construction operators build lists of particular values.
   b. The selection operators select the *nth* value of a list starting from 1, or the *mth* through *nth* values.
   c. The concatenation operator appends an element to the end of a list.
   d. The deletion operator removes the first occurrence (if any) of an element in a list.
   e. The membership operator returns true or false if an element is in a list.
   f. The length operator returns the number of elements in a list.
4. The tuple operators are used with values of tuple-type objects, i.e., objects declared with `'and'` composition.
   a. The construction operator builds a tuple of particular values.
   b. The selection operator selects a named tuple component.
5. The union operators are used with values of union-type objects, i.e., objects declared with `'or'` composition.
   a. The selection operator selects a named union component.
   b. The tag interrogation operator returns true if the value of a union-type is that of a specific named component.
6. The string operators are used with string-valued expressions, in the same manner as comparable operators in programming languages that support strings.
7. The relational operators have their standard meanings.
   a. Values of any type can be compared.
   b. Comparison of numeric values is standard.
   c. Comparison of boolean values maps to comparison of 0 for false and 1 for true.
   d. Comparison of string values is characterwise.
   e. Comparison of list values is elementwise, recursively to atomic types.
   f. Comparison of tuple values is componentwise, recursively to atomic types.
   g. Comparison of union values is based on the selected component value, recursively to atomic types.

D. Further details of the notation are covered in the FMSL reference manual, available online.

E. The logic of FMSL is comparable to other formal specification languages.

1. A difference between FMSL and a number of contemporary languages is the use in FMSL of lists instead of sets.
2. Formally, both lists and sets can be fully axiomatized (i.e., mathematically defined), so there is no lack of formality in the use of lists.
3. Overall, the use of lists instead of sets results in little difference in a specification.
   a. Set notation makes certain low-level specification easier than with lists, such as operations that can be modeled with set union and difference.
   b. On the other hand, list notation makes other forms of specification easier than with sets, such as specification of ordering constraints.

IV. "Programming" with predicates.

A. The language of predicates used in pre- and postconditions can be thought of as *non-procedural* programming.

B. The rules for this style of "programming" are different than the procedural kind.

1. We define data, but only in abstract terms and from an end-users "real world" perspective, not from a computer efficiency perspective.
2. We define functions, but only in abstract terms of what the functions do, not how they work.
3. Hence, the only "code" we have are boolean expressions at the beginning and ending of functions, no code bodies.
4. The closest thing we have to traditional control constructs are the two quantifiers `forall` and `exists`.
   a. However, these quantifiers are fundamentally different than normal programming language controls.
   b. Namely, they only return boolean values, and they don't make anything "happen".
5. Instead of procedural descriptions of how functions work (i.e., what happens *inside* a function), we have only true/false descriptions of what functions do (i.e., what's true *before* and *after* the function happens).
   a. Time does not pass within pre- or postconditions, even ones with quantifiers.
   b. Rather, pre- and postconditions are simply statements of mathematical fact, that are instantaneously true or false.
   c. Hence, even though a `forall` may seem somewhat like a for-loop, it is just a boolean expression that is only true or false.
   d. It may be a big boolean expression that is true in a lot of cases, but it's still just a boolean expression.

C. In some cases, it may be necessary to specify certain procedural aspects of a system, specifically the order in which operations occur.

1. However when we do this we need to be careful not to lapse into conventional programming.
2. Therefore, we will specify ordering constraints non-procedurally by writing the precondition of a successor operation to be dependent on the postcondition of a predecessor operation.
   a. E.g., if operation *B* must follow operation *A*, we write the postcondition of *A* such that the only way the precondition of *B* can be true is if *A*'s postcondition is true.
   b. In general, this is accomplished by having *A*'s postcondition require some unique value for one or more outputs, and then having *B*'s precondition state that its inputs must have the values required by *A*.
   c. In this way, we require that *A* must execute before *B*, if *B* is ever to happen.
3. As always, we will specify procedural (i.e., step-by-step) behavior only when it is *fundamental* to the way the user operates.
4. In particular, we need to be careful not to specify procedural details of a particular GUI, when it is only one particular way to access the abstract operations.
5. Here's the way to think about it -- if the user *must* perform a series of operations in a particular order, then we'll specify the order.

V. An initial example of fully formal specification.

A. For starters with pre- and postconditions, we'll begin with some Calendar tool operations that are simpler than the scheduling and viewing operations we've examined in recent weeks.

B. Specifically, we'll look at operations for adding and finding registered Calendar Tool users and groups.

C. These operations have useful but relatively straightforward specifications.

D. Next week we'll return to the specification of the more involved scheduling a viewing operations.

VI. Synopsis of requirements for user database admin functions.

A. When the user selects the '`Users ...`' item in the '`Admin`' menu, the system displays the screen shown in Figure 1.

1. User `Name` is a free-form string; `Id` is a unique system id of eight characters or less; `Email` address is free-form string; phone `Area` code is three digits, `Number` is seven digits;
2. The `Add` command adds a new user; `Id` field must be unique.
3. The `Find` command finds by `Name` or `Id` or both.
   a. If find is by name and the name is not unique, the system displays list of ids for users of that name.
   b. The user clicks on an item in the list to see the full record for that id.
   c. If no user of the given name or id is found, the system displays a "no users found" pop-up dialog.
4. `Change` works after the user changes the most recently displayed record.
   a. Typically, the user runs `Find` command first, then changes.
   b. The original record is removed, new record is added.
5. `Delete` removes the most recently displayed record, typically located with a `Find` command; the system displays an "are you sure" pop-up dialog for confirmation.

B. When the user selects the '`Groups ...`' item in the '`Admin`' menu, the system displays the screen in shown Figure 2.

1. Group `Name` is a free-form string that is unique for all groups; `leaders` and `Groups` are lists of user `Id`s for the group leaders and members, respectively; all leaders must be listed as members.
2. The `Add` command adds a new group; the `Name` must be unique.

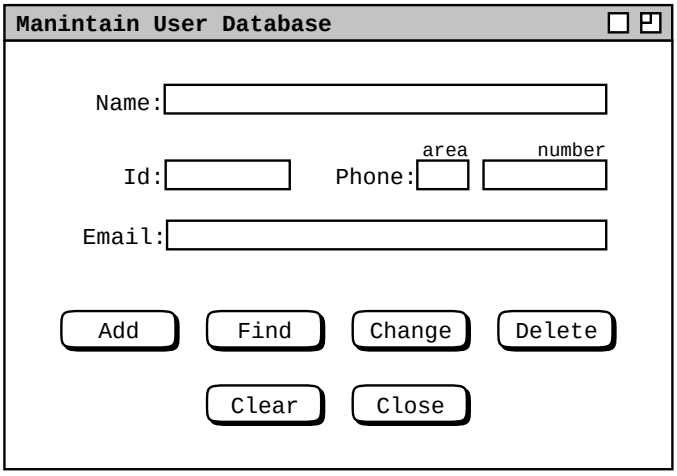


Figure 1: User database maintenance dialog.

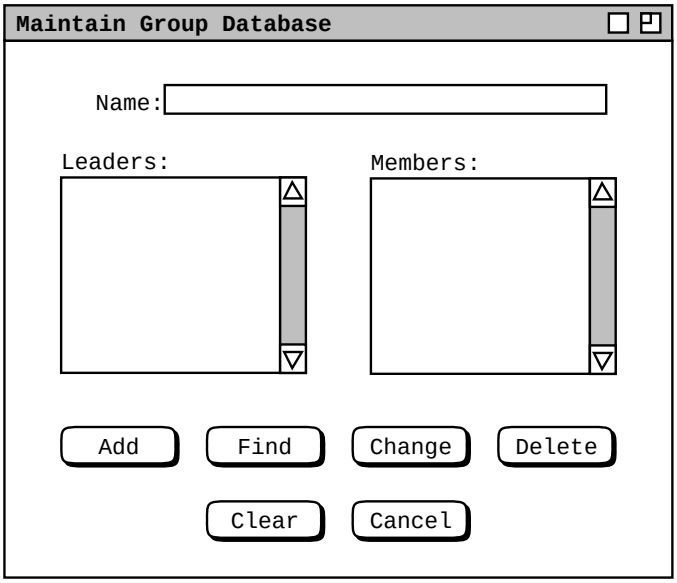


Figure 2: Group database maintenance dialog.

3. The `Find` command finds a group by name.
4. `Change` works after the user changes the most recently displayed group record.
   a. Typically, the user runs the `Find` command first, then changes.
   b. The original record is removed, the new record is added.
5. `Delete` removes the most recently displayed record, typically located with a `Find` command; the system displays an "are you sure" pop-up dialog for confirmation.

VII. Basic definitions for user database objects and operations.

A. Here are the relevant object and operation definitions:

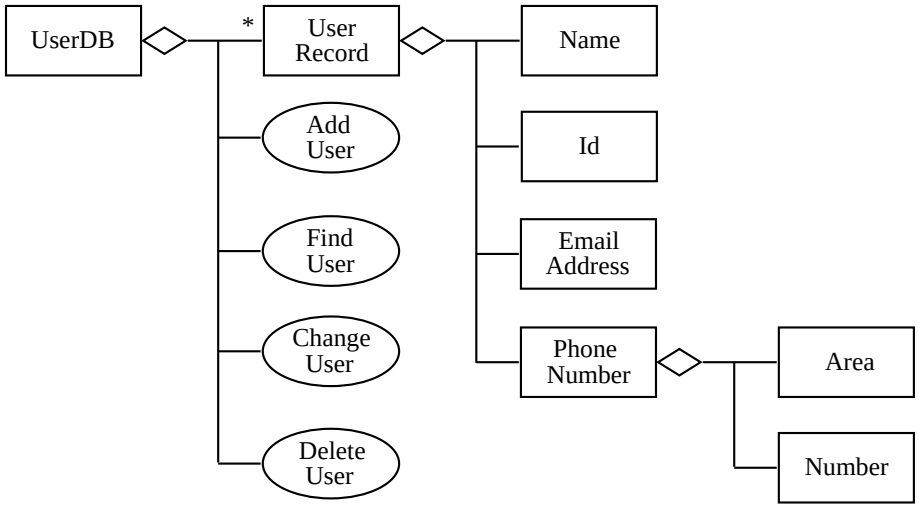
```
object UserDB
    components: UserRecord*;
    operations: AddUser, FindUser, ChangeUser, DeleteUser;
    description: (*
        UserDB is the repository of registered user information.
    *);
end UserDB;

object UserRecord
    components: Name and Id and EmailAddress and PhoneNumber;
    description: (*
        A UserRecord is the information stored about a registered user of the
        Calendar Tool.  The Name component is the user's real-world name.  The
        Id is the unique identifier by which the user is known to the Calendar
        Tool.  The EmailAddress is the electronic mail address used by the
        Calendar Tool to contact the user when necessary.  The PhoneNumber is
        for information purposes; it is not used by the Calendar Tool for
        contacting the user.
    *);
end User;

object Name = string;
object Id = string;
```

```
object EmailAddress = string;
object PhoneNumber = area:Area and number:Number;
object Area = integer;
object Number = integer;

operation AddUser
    inputs: UserDB, UserRecord;
    outputs: UserDB;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Add the given UserRecord to the given UserDB.  The Id of the given user
        record must not be the same as a user record already in the UserDB.
        The Id component is required and must be eight characters or less.  The
        email address is required.  The phone number is optional; if given, the
        area code and number must be 3 and 7 digits respectively.
    *);
end;

operation FindUser
    inputs: UserDB, Id;
    outputs: UserRecord;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Find a user by unique id.
    *);
end;

operation FindUser
    inputs: UserDB, Name;
    outputs: UserRecord;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Find a user or users by real-world name.  If more than one is found,
        output list is sorted by id.
    *);
end;

operation FindUser
    inputs: UserDB, Id, Name;
    outputs: UserRecord;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Find a user by both name and id.  This overload of FindUser is
        presumably used infrequently.  Its utility is to confirm that a
        particular user name and id are paired as assumed.
    *);
end;

operation ChangeUser
    inputs: UserDB, UserRecord, UserRecord;
    outputs:  UserDB;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Change the given old UserRecord to the given new record.  The old and
        new records must not be the same.  The old record must already be in
        the input db.  The new record must meet the same conditions as for the
        input to the AddUser operation.  Typically the user runs the FindUser
        operation prior to Change to locate an existing record to be changed.
    *);
end;
```

```
operation DeleteUser
    inputs: UserDB, UserRecord;
    outputs: UserDB;
    precondition: (* Coming soon *);
    postcondition: (* Coming soon *);
    description: (*
        Delete the given user record from the given UserDB.  The given record
        must already be in the input db.  Typically the user runs the FindUser
        operation prior to Delete to locate an existing record to delete.
    *);
end;
```

B. For a little practice with UML diagramming, Figure 3 shows diagrams for these definitions, in two equivalent formats.

One-part box format:



Equivalent three-part box format (with operation signature details):

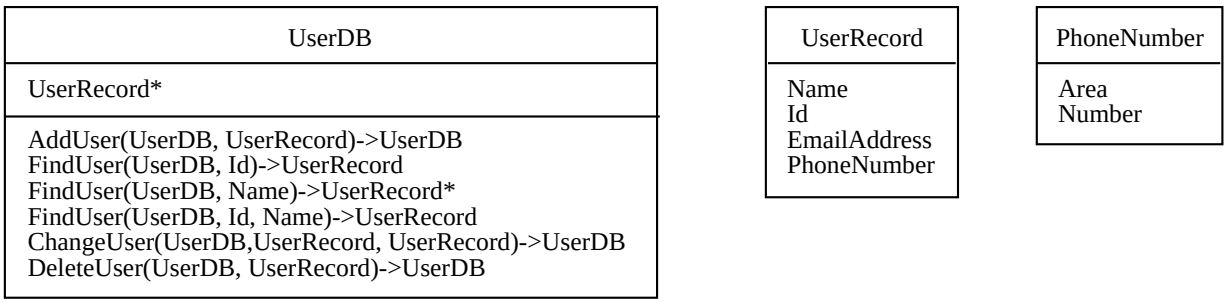


Figure 3: UML diagrams for UserDB objects and operations.

C. The objects and operations were derived from the user-level requirements, per the model derivation process discussed in Lecture Notes 5 last week.

D. The operation signatures are quite representative of those defined for a collection object.

1. `AddUser` is a *constructive* operation, with a signature of the general form

   `ConstructiveOp(Collection, Element)->Collection`

   with the effect of adding an element to the collection.

2. The versions of `FindUser` are *selective* operations, with signatures of the general form

   `SelectiveOp(Collection, UniqueElementSelector)->Element`
   `SelectiveOp(Collection, NonUniqueElementSelector)->Element*`

   with the effect of finding zero or more elements in a collection.

   a. In both forms, the second input is a component of *`Element`* used as a search key.
   b. In the first form, *`UniqueElementSelector`* is a component whose value is required to be unique among all elements of the collection.
   c. In the second form, *`NonUniqueElementSelector`* is a component whose value is not required to be unique among all elements of the collection.

3. `DeleteUser` is a *destructive* operation, with the same signature form as a constructive operation, but with the effect of removing rather than adding an element.

4. `ChangeUser` is a *modifying* operation (combined constructive and destructive), with a general signature of the form

   `EditingOp(Collection, OldElement, NewElement)->Collection`

   with the effect of removing the *`OldElement`* and adding the *`NewElement`*.

E. Note that the `FindUser` operation is *overloaded* with three definitions.

1. This is precisely the same kind of overloading supported by most modern programming languages.
2. Viz., an operation of the same name can be defined multiple times, as long as the input types are different in all definitions.
3. In terms of model accuracy, overloading works well in a case where the same operational widget (e.g., button) can be used with different input values.
4. In this case, for example, the user can type in either or both of the `Name` and `Id` fields of the dialog and use the same `Find` button to invoke the `FindUser` operation.
5. An alternative naming in lieu of overloading could be `FindUserById`, `FindUserByName`, and `FindUserByNameAndId`.
6. In this case, we'll say overloaded naming gets the nod in terms of modeling accuracy, though it's not a big deal.

VIII. An initial formal definition of `AddUser`.

A. For operation pre- and postconditions, we will start by stating a predicate in English, and then refine it into formal logic.

B. As we refine the logic, the English version will be retained as a comment, to aid in the human understanding of the specification.

C. So, here is an initial version of the formal spec for the AddUser operation:

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * The id of the given user record must be unique and less than or
         * equal to 8 characters; the email address must be non-empty; the
         * phone area code and number must be 3 and 7 digits, respectively.
```

```
         *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *);

    description: (* As above *);

end AddUser;
```

D. Before we get to the pre- and postcondition logic, we need to address one final notational matter -- naming inputs and outputs.

1. In the initial operation signatures presented above, only the *types* of inputs and outputs were specified, e.g.,

```
operation AddUser
    inputs: UserDB, UserRecord;
    outputs: UserDB;
```

2. This is sufficient for defining an operation until we get to the pre- and postconditions.
3. Within the conditions, the input and output objects must be uniquely referenced.
   a. Since in many cases input and output types may be the same, we need to give them some form of unique identifier.
   b. We do this by adding names as well as types, as described in the section on "Names and Types" in the specification language overview; e.g.

```
operation AddUser is
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
```

   c. Here the input names are *udb* for the `UserDB`, and *ur* for the `UserRecord`.
   d. The output name is *udb'*.
4. Input/output names in an operation definition serve essentially the same purpose as parameter names in a programming language function or procedure definition.
5. I.e., they provide the means to identify specific input and output objects by name.
6. There are two major syntactic differences between an FMSL operation signature and a comparable function or procedure declaration in a programming language.
   a. Unlike most programming languages, the single apostrophe character is legal in FMSL identifiers.
   b. By convention, if an operation uses the same type as both an input and output, the name of the output is the same as the input with an apostrophe appended; the apostrophe is read "prime".
   c. The other syntactic difference is the explicit naming of output objects.
      i. Most programming languages do not support multi-valued functions, and the output of a function is specified operationally with a `return` statement.
      ii. In an FMSL specification, the formal specification does not contain an operational "return", so the output object(s) must be explicitly named for reference purposes.
      iii. In addition, FMSL supports multi-valued operations, which require outputs to be distinguished by separate names.
7. Note also the FMSL comment syntax, which encloses comments in the brackets "`(*`" and "`*)`".

E. So, now we can get to some logic finally.

1. The English comment for the `AddUser` postcondition specifies the most fundamental property of an additive collection operation -- upon completion of the operation, the given element to be added is in the output collection.
2. Formally,

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition: (* Coming soon. *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        ur in udb';

end AddUser;
```

3. The simple expression "`ur in udb'`" is all there is to it.
   a. The `in` operator is built-in list membership.
   b. Its operands are a value of an element type and a value of a list type containing the element.
   c. In this case the operands are a `UserRecord` and a `UserDB`.

F. As it stands, `AddUser` still has no precondition formally defined, only a comment indicating what needs to be defined.
   1. Having no explicit precondition is equivalent to a precondition of true.
   2. In many cases, true preconditions are fine, given that there is no specific condition that must be met before the operation begins.
   3. In the case of the `AddUser` operation, a true precondition definitely won't do, since we can see from the requirements that a number of conditions must be met before `AddUser` can proceed.
   4. We will address these requirements step by step, as we refine the formal definition of `AddUser`.

IX. Refining the `AddUser` postcondition.

A. One of the fundamental questions that must be asked of pre- and postconditions is if they are *strong enough*.
   1. In general, adding additional predicate clauses strengthens the conditions.
   2. For example, the true precondition for `AddUser` is relatively weaker than one that specifies that there is no `UserRecord` of the same id already in the input database.

B. In general, there are two aims to strengthening a specification.
   1. Ensuring that all user-level requirements are met (cf. Maxim 1 above).
   2. Ensuring that a system implementation works properly (cf. Maxim 2).

C. The former is accomplished via continued consultation with the end user; the latter requires an experienced analyst, who understands the kinds of problems that may arise in a system implementation.

D. In the case of the user and group databases, as well as similar database applications, an area of potential implementation error is the introduction of spurious entries into the database and/or the spurious deletion of entries.

E. To avoid such spurious effects, the specification of `AddUser` is strengthened as follows:

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        (ur in udb')

            and
```

```
        (*
         * All the other records in the output db are those from the input db,
         * and only those.
         *)
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');

end AddUser;
```

F. This specification introduces the use of the universal quantification operator, `forall`.
   1. Universal quantification in FMSL has the same meaning as in standard (typed) predicate logic.
   2. The general format is the following:

      `forall` *`(x:t) predicate`*

   3. This is read "for all values *x* of type *t*, *predicate* is true" where *x* must appear somewhere in *predicate*.
   4. There are also two extended forms of `forall`, shown in Table 2.
   5. In general, universal quantification is used frequently when specifying predicates on list objects, as upcoming examples illustrate.

G. While this example is a good illustration of specification strengthening, there are easier ways to specify the same meaning logically.
   1. For example, the postcondition logic can be simplified to the following:

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * A user record is in the output db if and only if it is the new
         * record to be added or it is in the input db.
         *)
        forall (ur':UserRecord)
            (ur' in udb') iff ((ur' = ur) or (ur' in udb));

end AddUser;
```

   2. In general, predicate simplification is beneficial when it helps clarify the specification.
   3. Simplification is not necessary, as long as the logic is clear and accurate.

H. Another way to simplify this specification is to use a constructive list operator, as follows:

Extended Form	**Reading**	**Equivalent To**
`forall (`*`x:t | p1`*`) p2`	For all *x* of type *t*, such that *p1* is true, *p2* is true.	`forall (`*`x:t`*`) if `*`p1`*` then `*`p2`*
`forall (`*`x`*` in `*`l`*`) `*`p`*	For all *x* in *l*, *p* is true.	`forall (`*`x:basetype(l)`*`)` `  if `*`x`*` in `*`l`*` then `*`p`*

Table 2: Extended forms of universal quantification..

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        udb' = udb + ur;

end AddUser;
```

where '+' in this context is the list append operator.

1. A *constructive* specification such as this describes the output of an operation using a constructive operation on the inputs.
2. In contrast, an *analytic* specification (such as the previous spec using the `in` operation) describes output without using constructive operations.
3. In 308, we will define *analytic* specifications whenever possible.
   a. Specifically, we won't used the operations described as "construction" in Table 2.
   b. There is debate among software engineers as to the relative merits of constructive versus non-constructive specification; we will discuss the issues a bit later.

X. Refining the postconditions for the other `UserDB` operations.

A. Based on the development of the `AddUser` specs so far, we can provide a comparable level of formal specification for the other three `UserDB` operations.

B. For example, here is the idea for formalizing the `FindUser(UserDB,Id)` postcondition:

```
operation FindUser
    inputs: udb:UserDB, id:Id;
    outputs: ur':UserRecord;

    precondition: (* None yet. *);

    postcondition:
        (*
         * If there is a record with the given id in the input db, then the
         * output record is equal to that record, otherwise the output record
         * is empty.
         *);

    description: (*
        Find a user by unique id.
    *);
end FindUser;
```

C. In looking at the postcondition comment, we observe that the postcondition logic will need to refer to the id field of the input record, i.e., "a record with the given id".

1. This means that to formalize this operation, we must be able to refer to the `Id` component of a `UserRecord`.
2. This leads to a very common occurrence in the process of formalizing a specification.
3. Namely, we need to update the definition of an existing object, based on the need to specify a requirement precisely.
4. The update required here is to provide names for the components of the `UserRecord` object, so that the components can be individually referenced.

D. Here is the updated definition:

```
object UserRecord
    components: name:Name and id:Id and email:EmailAddress and
        phone:PhoneNumber;
    description: (*
        -- Same as before --
    *);
end UserRecord;
```

1. This definition uses name/type pairs in the same manner as in a full operation signature.
2. In a predicate, the named fields of a `UserRecord` can be referenced using the '.' operator, which has a comparable meaning to its use in a programming language when referencing the data fields of a class or struct.
3. That is, the compoents of an FMSL tuple define basically the same structure as the data fields in a programming language class.
4. The '.' operator is used to select a field of the tuple in the same way as it is used in Java or C++ to select the data field of a class.

E. With this update to the `UserRecord` object, here are the initial formal specifications for the `FindUser`, `ChangeUser`, and `DeleteUser` operations, with the "no spurious data" requirements.

```
operation FindUser
    inputs: udb:UserDB, id:Id;
    outputs: ur':UserRecord;

    precondition: (* None yet. *);

    postcondition:
        (*
         * If there is a record with the given id in the input db, then the
         * output record is equal to that record, otherwise the output record
         * is empty.
         *)
        (exists (ur in udb) (ur.id = id) and (ur' = ur))
            or
        (not (exists (ur in udb) (ur.id = id)) and (ur' = nil));

    description: (*
        Find a user by unique id.
    *);
end FindUser;

operation FindUser
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * A record is in the output list if and only it is in the input UserDB
         * and the record name equals the Name being searched for.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n));

    description: (*
        Find a user or users by real-world name.  If more than one is found,
        output list is sorted by id.
    *);
end FindUser;

operation FindUser
    inputs: udb:UserDB, id:Id, n:Name;
```

```
        outputs: ur':UserRecord;

        precondition: (* None yet. *);

        postcondition:
            (*
             * If there is a record with the given name and id in the input db,
             * then the output record is equal to that record, otherwise the output
             * record is empty.
             *)
            (exists (ur in udb) (ur.name = n) and (ur.id = id) and (ur' = ur))
                or
            (not (exists (ur in udb) (ur.name = n) and (ur.id = id)) and (ur' = nil));

        description: (*
            Find a user by both name and id.  This overload of FindUser is
            presumably used infrequently.  Its utility is to confirm that a
            particular user name and id are paired as assumed.
        *);
    end FindUser;

    operation ChangeUser
        inputs: udb:UserDB, old_ur:UserRecord, new_ur:UserRecord;
        outputs:  udb':UserDB;

        precondition: (* None yet. *);

        postcondition:
            (*
             * A user record is in the output db if and only if it is the new
             * record to be added or it is in the input db, and it is not the old
             * record.
             *)
            forall (ur':UserRecord)
                (ur' in udb') iff (((ur' = new_ur) or (ur' in udb)) and
                                   (ur' != old_ur));

        description: (*
            Change the given old UserRecord to the given new record.  The old and
            new records must not be the same.  The old record must already be in
            the input db.  The new record must meet the same conditions as for the
            input to the AddUser operation.  Typically the user runs the FindUser
            operation prior to Change to locate an existing record to be changed.
        *);
    end ChangeUser;

    operation DeleteUser
        inputs: udb:UserDB, ur:UserRecord;
        outputs: udb':UserDB;

        precondition: (* None yet. *);

        postcondition:
            (*
             * A user record is in the output db if and only if it is not the
             * existing record to be deleted and it is in the input db.
             *)
            (forall (ur':UserRecord)
                (ur' in udb') iff ((ur' != ur) and (ur' in udb)));

        description: (*
            Delete the given user record from the given UserDB.  The given record
            must already be in the input db.  Typically the user runs the FindUser
            operation prior to Delete to locate an existing record to delete.
        *);
    end DeleteUser;
```

F. Observations.

1. All of the preconditions are commented "`(* None yet. *)`"; we will refine preconditions shortly.
2. The postcondition for `FindUser(UserDB,Id)` uses the existential quantifier *`exists`*; Table 3 summarizes the formats.
3. The postcondition for `FindUser(UserDB,Name)` is missing an important piece of logic vis a vis user-level requirements. What is it?
4. The postcondition logic for `ChangeUser` and `DeleteUser` are adaptations of the postcondition logic for `AddUser`.
   a. This kind of logic is sometimes called the "no junk, no confusion" rule for collection classes.
   b. Namely, when we put something into or take something out of a collection,
      i. we don't put in or take out anything superfluous (no junk),
      ii. we do put in or take out exactly what we intend to (no confusion).
   c. You should study the logic closely to clarify your understanding of it.

XI. On the use of quantifiers.

A. Universal and existential quantification are two ways to state mulitple conditions in a single expression.

1. With universal quantification (`forall`), the quantifier expression is true if *all* cases considered are true.
2. With existential quantification (`exists`), the quantifier expression is true if *at least one* of the cases is true.
3. Logically, you can think of `forall` and `exists` as forms of repeated `and` and `or`, respectively.
4. There is even a generalized DeMorgan's law that makes the two forms of quantifier interchangeable:

```
forall (x:t) p <=> not (exists (x:t) not p)
    and
not (forall (x:t) not p) <=> exists (x:t) p
```

B. In the software modeling task upon which we're focused, the use of logical quantifiers is focused on two specific objectives:

1. Stating a requirement about all values of a particular type, e.g.,

   `forall (ur:UserRecord)` *`requirement-predicate`*

2. Stating a requirement that must be true for at least one value of a particular type, e.g.,

Form	**Reading**	**Equivalent To**
`exists (`*`x:t`*`)` *`p`*	There exists *x* of type *t* such that predicate *p* is true.	
`exists (`*`x:t | p1`*`)` *`p2`*	There exists *x* of type *t*, such that *p1* is true and *p2* is true.	`exists (`*`x:t`*`)` *`p1`* `and` *`p2`*
`exists (`*`x`* `in` *`l`*`)` *`p`*	There exists *x* in *l* such that *p* is true.	`exists (`*`x:basetype(s)`*`) (`*`x`* `in` *`s`*`) and` *`p`*

Table 3: Forms of existential quantification..

```
exists (ur:UserRecord) requirement-predicate
```

C. The specialized forms of qualification provide further focus.

1. Stating a requirement about all values (or at least one value) in a particular list, e.g.,

```
forall (ur in udb) requirement-predicate
exists (ur in udb) requirement-predicate
```

2. Stating a requirement about all values (or at least one value) of a particular type, with some further restrictions on the values. E.g.,

```
forall (i:integer | i > 0) requirement-predicate
exists (i:integer | i > 0) requirement-predicate
```

D. Keeping these specific focuses in mind will help narrow down when and how to use quantifiers.

XII. Formally specifying user-level requirements.

A. To this point, we have formalized some basic requirements for our database operations.

B. Specifically, we have focused on postconditions related to the second of our formal specification maxims -- not trusting the programmer.

C. It is now time to consider the formal definition of user-level requirements per the first maxim -- *nothing is obvious*.

D. To start, there are a number of "obvious" user-level requirements, including the following:

1. Duplicate entries are not allowed in the `UserDB`.
2. Input values are checked for validity.
3. If the `FindUser` operation outputs more than one record, the output should be sorted in some appropriate order.

E. We have considered these requirements to some extent in the requirements narrative.

1. However, the process of fully formalizing the specification can reveal important details we may have overlooked in the requirements scenarios.
2. For example, in the Milestone 6 scenarions we initially overlooked the sorting requirement for multiple outputs from `FindUser`.
3. Such oversights are common, and one of the main reasons we're doing the fully formal level of the spec.

F. An historical note is of interest with regards to such requirements.

1. In software engineering methodologies less formal than what we're using, the process of formalizing a specification can take the form of "firming up" the English prose in which the requirements are stated.
2. For example, the first of the above requirements could be stated "formally" as follows:

   A UserDB shall not contain duplicate entries.

3. While this may not seem to be a substantial improvement to the original statement of the requirement, it represents a seriously-proposed approach to formalization.
   a. With this approach, a number of possible forms of natural language are standardized with a restricted vocabulary.
   b. For example, all formal requirements are expressed using "shall" instead of other comparable English words such as "should", "ought to", or "allowed to".
4. This idea of formalizing English is noteworthy because it has been widely used in practice, and significant documents have been "formalized" in this manner.
5. While such rules can indeed help with the formalization process, they fall well short of a fully formal basis for requirements specification.

XIII. No Duplicates

A. Analysis of the no duplicates requirement provides fine support for the "nothing-is-obvious" maxim.

B. While we may expect reasonable people to understand what "no duplicates" means, there are in fact a number of plausible interpretations here.

C. Three such interpretations are the following:

1. No two `UserRecord`s in a `UserDB` have exactly the same values for all `UserRecord` components.
2. No two `UserRecord`s in a `UserDB` have the same name.
3. No two `UserRecord`s in a `UserDB` have the same id.

D. Which of these interpretations to choose is categorically *not* a matter for a programmer to decide.

1. Rather, it should be decided at the user specification level, by the analyst in consultation with the end users.
2. We could even grant that most programmers are reasonably smart, so in this case we might safely assume that a programmer could make the correct decision, or know enough to consult with the user to resolve the ambiguity.
3. Suppose, however, we were specifying data records in a much more complicated application domain, such as aeronautics.
4. In this domain we might have a data object such as an anomaly list, with record fields like `PreFlight`, `TaxiOut`, `InFlight`, `Approach`, and `Landing`.
   a. What does it mean to disallow duplicates in an anomalies database?
   b. Which field, if any, could be used as a unique key?
5. The point is that such questions need to be answered by end users and/or application domain experts.
6. Such questions should most certainly not be left unanswered when the programmer begins work, since the programmer may well not know how to answer them, or even that they need to be asked.

E. In our `UserDB` case, we have already determined with the customer that the `Id` component of a `UserRecord` is the unique key.

1. This means that `UserRecord`s in the `UserDB` need only differ in the Id value.
2. In particular, there may be multiple `UserRecord`s with the same name.

F. The basic strategy for disallowing duplicates is to define a precondition on `AddUser` that checks for an element of the same Id as the `UserRecord` being added.

G. Here is the refined specification for `AddUser`; for brevity, the postcondition is omitted:

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same id as the
         * record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id));


    postcondition: (* Same as above *);

end AddUser;
```

H. A discussion of the exact nature of a precondition is in order here.

1. By definition, failure of a precondition means that the operation is prevented from executing.
2. More precisely, precondition failure means that the operation fails and produces a value of nil.

3. The nil value in FMSL is defined for all object types, and is a distinct value from any other value of a given type.
4. This abstract meaning of precondition failure does not define how operation failure is perceived by the end user.
   a. Generally, the end-user should see an appropriate error message when an operation fails.
   b. The details of such error messages are typically abstracted out of the formal specification.

XIV. Input value checking.

A. Input value constraints for a user record are described in the requirements scenarios as follows:
   1. the Id of a user record is a unique system id of eight characters or less;
   2. the email address is a free-form string;
   3. the phone area code is three digits, the number is seven digits.

B. These constraints are defined formally as follows, with accompanying commentary:

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * There is no user record in the input UserDB with the same id as the
         * record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))

            and

        (*
         * The id of the given user record is not empty and 8 characters or
         * less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)

            and

        (*
         * The email address is not empty.
         *)
        (ur.email != nil)

            and

        (*
         * If the phone area code and number are present, they must be 3 digits
         * and 7 digits respectively.
         *)
        (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
        (if (ur.phone.num != nil) then (#(ur.phone.num) = 7));


    postcondition: (* Same as above *);

end AddUser;
```

C. Observations
   1. The standard way to strengthen a precondition is to `and` on additional clauses.
      a. Here, the previous "no duplicates" clause remains.
      b. The new requirements are added by `and`ing them on.

2. The process of formally specifying these requirements led to the discovery of one unnoticed requirements detail, which will be updated in the scenario narrative.
3. Specifically, in considering the formal specification for the constraint on email address, we were alerted to the question of whether it should be required.
   a. In consultation with the customer, the answer turns out to be "yes", even though we had not originally considered the issue explicitly in the scenarios.
   b. Hence, there is the precondition clause

```
(ur.email != nil)
```

   c. This says that while the email address can be a free-form string of any length, it cannot be nil, i.e., the user cannot leave it empty in the dialog.
   d. Such are just the kind of details we hope to catch while formalizing.

XV. Ordering of multi-record output lists.

A. The version of `FindUser` with the `Name` input produces a list of `UserRecord`s, since `Name` is not required to be a unique-valued component of a record.

B. As noted above, the initial requirements scenario overlooked what order the outputs should be in if there are two or more.

C. The most reasonable choice is to sort the output list by Id field.
   1. The scenario narrative will be updated to reflect this decision.
   2. As with other such requirements, we should not trust that a programmer will do the right thing in the absence of a formal statement.
   3. In this case, the programmer may not even think there is problem if an output list is displayed in some internal order, such as the order `UserRecord`s are stored in a hash table.
   4. Such an order is as good as random to most human users, and as such not satisfactory.

D. To specify `UserRecord` list ordering, we must strengthen the `FindUser` postcondition. Here it is:

```
operation FindUser
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name in the
         * input db.
         *)
        (forall (ur: UserRecord) (ur in url) iff (ur in udb) and (ur.name = n))

            and

        (*
         * The output list is sorted alphabetically by id.
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
            url[i].id < url[i+1].id);

    description: (*
        Find a user or users by real-world name.  If more than one is found,
        the output list is sorted by id.
    *);
end FindUser;
```

E. An English translation of the sorting logic is the following:

"For each position i in the output list, such that i is between the first and the second to the last positions in the list, the ith element of the list is less than the i+1st element of the list."

F. You should study this logic to be satisfied that it specifies sorting satisfactorily.

XVI. Unbounded quantification.

A. What would happen to the meaning of the sorting predicate if the restrictions on the range of i were not present?

B. I.e., if the sorting logic in the postcondition were changed to the following:

```
forall(i: integer)
    url[i].id < url[i+1].id);
```

C. The meaning here is an *unbounded quantification*.

1. That is, the quantifier operates over the infinite range of all integers.
2. In principle, there is nothing wrong with unbounded quantification.
3. For example, the original anti-spurious requirements for `AddUser` are expressed using unbounded quantification.
4. One might argue for range restrictions on the grounds of efficiency, but as noted earlier, efficiency of this nature is not of concern in an abstract specification.

D. The potential problem with unbounded quantification is that the body of the universal quantifier may not have the correct value in an unbounded range, and hence the value of the entire quantifier expression may be false when we expect it to be true.

1. This is in fact the case in the unbounded quantification used in the sorting predicate for `Find-User(UserDB,Name)`.
2. Here is specifically what goes wrong in the unbounded quantifier version of the sorting logic:
   a. When `i` is outside of the range of `[1..#url]`, then the value of the expression `url[i]` is `nil`.
   b. This is the case by the language rules of FMSL, that define the value of an index expression to be `nil` if the value of the index is outside of the bounds of the indexed list.
   c. When the value of `url[i]` is `nil`, the value of `url[i].id` goes to `nil`.
   d. This again is by the rules of FMSL, that define the value of a selection expression (containing '.') to be `nil` if the value of the object being selected from is `nil`.
   e. This in turn leads to the following expression as the body of the forall:

      ```
      nil < url[i+1].id
      ```

      the value of which is false.
   f. This result is due to the FMSL rule that defines the value of '<' to be false if either or both of its operands is `nil`.
   g. Finally, any value of false in the body of the forall drives the value of the entire forall to false, by the normal rules of forall.
   h. This means that the postcondition is *always false*, even if the output is properly sorted.

E. To some extent, this exact outcome of the unbounded quantification is due to the particular semantic rules of FMSL.

1. In general, however, unbounded quantification is potentially problematic under any logical semantics.
2. The point is that one needs to be careful when using unbounded quantification to ensure that the body of the quantifier has a well understood value over the entire unbounded range of quantification.
3. This is particularly the case when quantifying over the elements of a list.

XVII. Using auxiliary functions.

A. The postcondition in the most recent definition of `FindUser` is a little lengthy.
   1. In practice, predicates significantly longer than this can appear in the specification of a complex operation.
   2. When pre- or postconditions become unduly long, it is useful to use *auxiliary functions* to organize the logic.
   3. An auxiliary function is much the same as a function definition in a programming language, with the restrictions of functional semantics we have been observing.
   4. The purpose of an auxiliary function is modularize a piece of logic, give it a mnemonic name, and allow that logic to be invoked in one or more places.

B. As an example, here is the last definition of `FindUser` using two auxiliary functions.

```
operation FindUser
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    postcondition:
        RecordsFound(udb,n,url)
            and
        SortedById(url);

end FindUser;

function RecordsFound(udb:UserDB, n:Name, url:UserRecord*) =
    (*
     * The output list consists of all records of the given name in the input
     * db.
     *)
    (forall (ur' in url) (ur' in udb) and (ur'.name = n));

function SortedById(url:UserRecord*) =
    (*
     * The output list is sorted alphabetically by id.
     *)
    (forall (i:integer | (i >= 1) and (i < #url))
        url[i].id < url[i+1].id);
```

   1. Semantically, there is no difference between an auxiliary function and an operation.
   2. They both define objects of a function type.
   3. In fact, the keyword "operation" can be used in place of the keyword "function".
      a. The separate keywords are provided to suggest different usages within a specification.
      b. By convention the keyword "operation" should be used to define an operation that is directly visible to the end user, i.e., an operation that can be traced directly to some user interface element.
      c. In contrast, the keyword "function" should be used for functions that are not normally visible to the user, but which are used solely to clarify the logic of a formal specification.

XVIII. Moving on to the specs for the GroupDB.

A. Figure 2 on page 5 shows the UI for the other user-related database in the Calendar Tool -- the database of user groups.

B. The specs for the GroupDB are quite similar to UserDB.
   1. Both databases are clear examples of collection objects with typical collection operations.
   2. The specs for GroupDB are slightly simpler, given that there is only one searchable component, the group name, which must be unique among all groups in the database.

C. A significant specification issue does arise in the area of interaction between user database operations with the group database.

1. Specifically, what happens to groups that have as a member a user who is deleted from the user database?
2. Possible ways to deal with this problem include the following:
   a. A deleted user is automatically removed from all groups of which she is a member.
   b. If a deleted user is in one or more groups, a warning message is output indicating what groups the user was in, but the users must be manually deleted from the groups; in the meantime, any unknown users are simply ignored in the group member lists.
   c. The system prevents deletion of a user until she has first been deleted from all groups; to assist the deletion, the system outputs a message indicating the affected groups.

D. This is yet another example of where formalizing the specs has led to the discovery of an important requirements issue.
   1. In this case, user consultation results in the automatic removal solution.
   2. This in turn leads to another issue, which is what should be done with groups who have no leader, due to the automatic deletion of a member or was the only leader of a group.
   3. This issue is resolved by allowing leaderless groups, but having the system output a warning when the situation arises.

E. All of the issues having been resolved, the resulting complete spec for the user and group databases is as follows:

```
(****
 *
 * Module Admin defines the objects and operations related to maintaining the
 * user, group, location, and global options databases of the Calendar Tool.
 *
 *)
module Admin;

  export UserDB, GroupDB, LocationDB, UserId;

  object UserDB
    components: UserRecord*;
    description: (*
        UserDB is the repository of registered user information.  It is a
        collection of UserRecords.
    *);
  end UserDB;

  object UserRecord
    components: name:Name and id:UserId and email:EmailAddress and
        phone:PhoneNumber;
    description: (*
        A UserRecord is the information stored about a registered user of the
        Calendar Tool.  The Name component is the user's real-world name.  The
        UserId is the unique identifier by which the user is known to the
        Calendar Tool.  The EmailAddress is the electronic mail address used by
        the Calendar Tool to contact the user when necessary.  The PhoneNumber
        is for information purposes; it is not used by the Calendar Tool for
        contacting the user.
    *);
  end UserRecord;

  object Name = string;
  object UserId = string;
  object EmailAddress = string;
  object PhoneNumber = area:Area and number:Number;
  object Area = integer;
  object Number = integer;

  operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
```

```
  outputs: udb':UserDB;
  description: (*
      Add the given UserRecord to the given UserDB.  The UserId of the given
      user record must not be the same as a user record already in the
      UserDB.  The UserId component is required and must be eight characters
      or less.  The email address is required.  The phone number is optional;
      if given, the area code and number must be 3 and 7 digits respectively.
  *);

  precondition:
      (*
       * There is no user record in the input UserDB with the same id as the
       * record to be added.
       *)
      (not (exists (ur' in udb) ur'.id = ur.id))

          and

      (*
       * The id of the given user record is not empty and 8 characters or
       * less.
       *)
      (ur.id != nil) and (#(ur.id) <= 8)

          and

      (*
       * The email address is not empty.
       *)
      (ur.email != nil)

          and

      (*
       * If the phone area code and number are present, they must be 3 digits
       * and 7 digits respectively.
       *)
      (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
      (if (ur.phone.number != nil) then (#(ur.phone.number) = 7));

  postcondition:
      (*
       * A user record is in the output db if and only if it is the new
       * record to be added or it is in the input db.
       *)
      forall (ur':UserRecord)
          (ur' in udb') iff ((ur' = ur) or (ur' in udb));

end AddUser;

operation FindUser
  inputs: udb:UserDB, id:UserId;
  outputs: ur':UserRecord;
  description: (*
      Find a user by unique id.
  *);

  precondition: ;

  postcondition:
      (*
       * If there is a record with the given id in the input db, then the
       * output record is equal to that record, otherwise the output record
       * is empty.
       *)
      (exists (ur in udb) (ur.id = id) and (ur' = ur))
```

```
          or
      (not (exists (ur in udb) (ur.id = id)) and (ur' = nil));

end FindUser;

operation FindUser
  inputs: udb:UserDB, n:Name;
  outputs: url:UserRecord*;
  description: (*
      Find a user or users by real-world name.  If more than one is found,
      the output list is sorted by id.
  *);

  precondition: ;

  postcondition:
      (*
       * The output list consists of all records of the given name in the
       * input db.
       *)
      (forall (ur' in url) (ur' in udb) and (ur'.name = n))

          and

      (*
       * The output list is sorted alphabetically by id.
       *)
      (forall (i:integer | (i >= 1) and (i < #url))
          url[i].id < url[i+1].id);

end FindUser;

operation FindUser
  inputs: udb:UserDB, id:UserId, n:Name;
  outputs: ur':UserRecord;
  description: (*
      Find a user by both name and id.  This overload of FindUser is
      presumably used infrequently.  Its utility is to confirm that a
      particular user name and id are paired as assumed.
  *);

  precondition: ;

  postcondition:
      (*
       * If there is a record with the given name and id in the input db,
       * then the output record is equal to that record, otherwise the output
       * record is empty.
       *)
      (exists (ur in udb) (ur.name = n) and (ur.id = id) and (ur' = ur))
          or
      (not (exists (ur in udb) (ur.name = n) and (ur.id = id)) and
              (ur' = nil));

end FindUser;

operation ChangeUser
  inputs: udb:UserDB, gdb:GroupDB, old_ur:UserRecord, new_ur:UserRecord;
  outputs:  udb':UserDB, gdb':GroupDB;
  description: (*
      Change the given old UserRecord to the given new record.  The old and
      new records must not be the same.  The old record must already be in
      the input db.  The new record must meet the same conditions as for the
      input to the AddUser operation.  Typically the user runs the FindUser
      operation prior to Change to locate an existing record to be changed.
                                                                          <p>
```

```
    If the user record id is changed, then change all occurrences of the
    old id in the group db to the new id.
*);

precondition:
    (*
     * The old and new user records are not the same.
     *)
    (old_ur != new_ur)

        and

    (*
     * The old record is in the given db.
     *)
    (old_ur in udb)

        and

    (*
     * There is no user record in the input UserDB with the same id as the
     * new record to be added.
     *)
    (not (exists (new_ur' in udb) new_ur'.id = new_ur.id))

        and

    (*
     * The id of the new record is not empty and 8 characters or less.
     *)
    (new_ur.id != nil) and (#(new_ur.id) <= 8)

        and

    (*
     * The email address is not empty.
     *)
    (new_ur.email != nil)

        and

    (*
     * If the phone area code and number are present, they must be 3 digits
     * and 7 digits respectively.
     *)
    (if (new_ur.phone.area != nil) then (#(new_ur.phone.area) = 3)) and
    (if (new_ur.phone.number != nil) then (#(new_ur.phone.number) = 7));

postcondition:
    (*
     * A user record is in the output db if and only if it is the new
     * record to be added or it is in the input db, and it is not the old
     * record.
     *)
    forall (ur':UserRecord)
        (ur' in udb') iff (((ur' = new_ur) or (ur' in udb)) and
                           (ur' != old_ur))

        and

    (*
     * If new id is different than old id, then all occurrences of old id
     * in the GroupDB are replaced by new id.
     *)
    if (old_ur.id != new_ur.id)
    then
```

```
          ... (* Logic left as exercise for the reader. *) ;

end ChangeUser;

operation DeleteUser
  inputs: udb:UserDB, gdb:GroupDB, ur:UserRecord;
  outputs: udb':UserDB, gdb':GroupDB, lgw:LeaderlessGroupsWarning;
  description: (*
      Delete the given user record from the given UserDB.  The given record
      must already be in the input db.  Typically the user runs the FindUser
      operation prior to Delete to locate an existing record to delete.
                                                                        <p>
      In addition, delete the user from all groups of which the user is a
      member.  If the deleted user is the only leader of a one more groups,
      output a warning indicating that those groups have become leaderless.
  *);

  precondition:
      (*
       * The given UserRecord is in the given UserDB.
       *)
      ur in udb;

  postcondition:
      (*
       * A user record is in the output db if and only if it is not the
       * existing record to be deleted and it is in the input db.
       *)
      (forall (ur':UserRecord)
          (ur' in udb') iff ((ur' != ur) and (ur' in udb)))

          and

      (*
       * The id of the deleted user is not in the leader or member lists of
       * any group in the output GroupDB.  (NOTE: This clause is not as
       * strong as a complete "no junk, no confusion" spec.  Why not?  Should
       * it be?)
       *)
      (forall (gr in gdb')
          (not (ur.id in gr.leaders)) and (not (ur.id in gr.members)))

          and

      (*
       * The LeaderlessGroupsWarning list contains the ids of all groups
       * whose only leader was the user who has just been deleted.
       *)
      (forall (gr in gdb)
          forall (id:UserId)
              (id in lgw) iff ((#(gr.leaders) = 1) and
                               (gr.leaders[1] = ur.id)));

end DeleteUser;

object LeaderlessGroupsWarning
  components: Name*;
  description: (*
      LeaderlessGroupsWarning is an secondary output of the Change and
      DeleteUser operations, indicating the names of zero or more groups that
      have become leaderless as the result of a user having been deleted.
  *);
end LeaderlessGroupsWarning;

object GroupDB
  components: GroupRecord*;
```

```
  description: (*
      UserDB is the repository of user group information.
  *);
end GroupDB;

object GroupRecord
  components: name:Name and leaders:Leaders and members:Members;
  description: (*
      A GroupRecord is the information stored about a user group.  The Name
      component is a unique group name of any length.  Leaders is a list of
      zero or more users designated as group leader.  Members is the list of
      group members, including the leaders.  Both lists consist of user id's.
      Normally a group is required to have at least one leader.  The only
      case that a group becomes leaderless is when its leader is deleted as a
      registered user.
  *);
end GroupRecord;

object Leaders = UserId*;
object Members = UserId*;

operation AddGroup
  inputs: gdb:GroupDB, udb:UserDB, gr:GroupRecord;
  outputs: gdb':GroupDB;
  description: (*
      Add the given GroupRecord to the given GroupDB.  The name of the given
      group must not be the same as a group already in the GroupDB.  All
      group members must be registered users.  The leader(s) of the group
      must be members of it.
  *);

  precondition:
      (*
       * All group members are registered users.
       *)
      (forall (id in gr.members) exists (ur in udb) ur.id = id)

          and

      (*
       * All group leaders are members of the group.
       *)
      (forall (id in gr.leaders) id in gr.members);

  postcondition:
      (*
       * A group record is in the output db if and only if it is the new
       * record to be added or it is in the input db.
       *)
      forall (gr':GroupRecord)
          (gr' in gdb') iff ((gr' = gr) or (gr' in gdb));

end AddGroup;

operation FindGroup
  inputs: gdb:GroupDB, n:Name;
  outputs: gr':GroupRecord;
  description: (*
      Find a group by unique name.
  *);

  precondition: ;

  postcondition:
      (*
       * If there is a record with the given name in the input db, then the
```

```
         * output record is equal to that record, otherwise the output record
         * is empty.
         *)
        (exists (gr in gdb) (gr.name = n) and (gr' = gr))
            or
        (not (exists (gr in gdb) (gr.name = n)) and (gr' = nil));

end FindGroup;

operation ChangeGroup
  inputs: gdb:GroupDB, udb:UserDB, old_gr:GroupRecord, new_gr:GroupRecord;
  outputs: gdb':GroupDB;
  description: (*
      Change the given old GroupRecord to the given new record.  The old and
      new records must not be the same.  The old record must already be in
      the input db.  The new record must meet the same conditions as for the
      input to the AddGroup operation.  Typically the user runs the FindGroup
      operation prior to Change to locate an existing record to be changed.
  *);

  precondition:
      (*
       * The old and new group records are not the same.
       *)
      (old_gr != new_gr)

          and

      (*
       * All group members are registered users.
       *)
      (forall (id in new_gr.members) exists (ur in udb) ur.id = id)

          and

      (*
       * All group leaders are members of the group.
       *)
      (forall (id in new_gr.leaders) id in new_gr.members);

  postcondition:
      (*
       * A group record is in the output db if and only if it is the new
       * record to be added or it is in the input db, and it is not the old
       * record.
       *)
      forall (gr':GroupRecord)
          (gr' in gdb') iff (((gr' = new_gr) or (gr' in gdb)) and
                             (gr' != old_gr));

end ChangeGroup;

operation DeleteGroup
  inputs: gdb:GroupDB, gr:GroupRecord;
  outputs: gdb':GroupDB;
  description: (*
      Delete the given group record from the given GroupDB.  The given record
      must already be in the input db.  Typically the user runs the FindGroup
      operation prior to Delete to locate an existing record to delete.
  *);

  precondition:
      (*
       * The given GroupRecord is in the given GroupDB.
       *)
      gr in gdb;
```

```
      postcondition:
          (*
           * A group record is in the output db if and only if it is not the
           * existing record to be deleted and it is in the input db.
           *)
          (forall (gr':GroupRecord)
              (gr' in gdb') iff ((gr' != gr) and (gr' in gdb)));

    end DeleteGroup;

    object LocationDB
      components: LocationRecord*;
      description: (*

      *);
    end LocationDB;

    object LocationRecord
      components: ;
      description: (*

      *);
    end LocationRecord;


    (*
     * Global options TBD.
     *)

  end Admin;
```