CSC 307 Lecture Notes Week 5

Introduction to Formal Specification

I. Practical Benefits of Formal Specification

I. Practical Benefits of Formal Specification

A. Better understanding of software.

I. Practical Benefits of Formal Specification

A. Better understanding of software.

B. Precise communication among developers.

I. Practical Benefits of Formal Specification

A. Better understanding of software.

B. Precise communication among developers.

C. Basis for thorough (maybe automated) testing.

I. Practical Benefits of Formal Specification

A. Better understanding of software.

B. Precise communication among developers.

C. Basis for thorough (maybe automated) testing.

D. Basis for verification (when appropriate).

I. Practical Benefits of Formal Specification

A. Better understanding of software.

B. Precise communication among developers.

C. Basis for thorough (maybe automated) testing.

D. Basis for verification (when appropriate).

E. Basis for automatic programming (in future).

F. Motivational Bottom Line

1. Suppose your boss says:

"I want you to do whatever it takes to build me software of the best possible quality, that has the smallest possible likelihood of failing."

F. Motivational Bottom Line

1. Suppose your boss says:

"I want you to do whatever it takes to build me software of the best possible quality, that has the smallest possible likelihood of failing."

2. For many, formal specification is a key part of responding to a mandate like this.

II. What’s Involved

II. What’s Involved

A. Formalize model as operations become well defined.

II. What’s Involved

A. Formalize model as operations become well defined.

B. The 307 technique is based on operation *preconditions* and *postconditions*.

Pre- and postconditions, cont’d

1. Precondition is true before op executes.

Pre- and postconditions, cont’d

1. Precondition is true before op executes.

2. Postcondition is true after completion.

Pre- and postconditions, cont’d

1. Precondition is true before op executes.

2. Postcondition is true after completion.

3. This specification style called *predicative*.

Pre- and postconditions, cont’d

1. Precondition is true before op executes.

2. Postcondition is true after completion.

3. This specification style called *predicative*.

4. A *predicate* is just a boolean expression, which is what pre- and postconditions are.

Pre- and postconditions, cont’d

C. Conditions specify fully what system does, including all user-level requirements.

Pre- and postconditions, cont'd

C. Conditions specify fully what system does, including all user-level requirements.

D. Formalizing specs is part of the following requirements/specification process:

Pre- and postconditions, cont’d

1. write user-level scenarios

Pre- and postconditions, cont’d

1. write user-level scenarios

2. model objects and operations

Pre- and postconditions, cont’d

1. write user-level scenarios
2. model objects and operations
3. formalize operations

Pre- and postconditions, cont’d

1. write user-level scenarios
2. model objects and operations
3. formalize operations
4. refine user-level scenarios

Pre- and postconditions, cont’d

1. write user-level scenarios

2. model objects and operations

3. formalize operations

4. refine user-level scenarios

5. refine formalized model

Pre- and postconditions, cont’d

1. write user-level scenarios
2. model objects and operations
3. formalize operations
4. refine user-level scenarios
5. refine formalized model
6. iterate 1-5 until done

Pre- and postconditions, cont’d

E. "Until done" involves two levels of validation:

Pre- and postconditions, cont’d

E. "Until done" involves two levels of validation:

1. Complete, consistent for end users.

Pre- and postconditions, cont’d

E. "Until done" involves two levels of validation:

1. Complete, consistent for end users.

2. Complete, consistent formally.

III. Formal specification maxims

III. Formal specification maxims

A. Observe:

III. Formal specification maxims

A. Observe:

1. Nothing is obvious.

III. Formal specification maxims

A. Observe:

1. Nothing is obvious.

2. Never trust the programmer.

Maxims, cont’d

B. First maxim relates to user-level requirements.

Maxims, cont’d

B. First maxim relates to user-level requirements.

C. Second maxim avoids nasty surprises in an implementation.

IV. Spest predicate notation

IV. Spest predicate notation

A. Variant of mathematical logic.

IV. Spest predicate notation

A. Variant of mathematical logic.

B. Includes augmented boolean expressions, arithmetic, collections, strings.

IV. Spest predicate notation

A. Variant of mathematical logic.

B. Includes augmented boolean expressions, arithmetic, collections, strings.

C. Summarized in Table 1.

Predicate Logic:

Predicate Logic:

Operator	Description

Predicate Logic:

Operator	Description
&&	logical and

Predicate Logic:

Operator	Description
&&	logical and
||	logical or

Predicate Logic:

Operator	Description
&&	logical and
||	logical or
!	logical not

Predicate Logic:

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>||</td><td>logical or</td></tr>
<tr><td>!</td><td>logical not</td></tr>
<tr><td>if (...)</td><td>logical implication</td></tr>
</table>

Predicate Logic:

Operator	Description
&&	logical and
||	logical or
!	logical not
if (...)	logical implication
if (...) else	logical choice

Predicate Logic:

Operator	Description
&&	logical and
||	logical or
!	logical not
if (...)	logical implication
if (...) else	logical choice
iff	logical equivalence

Predicate Logic:

Operator	Description
&&	logical and
||	logical or
!	logical not
if (...)	logical implication
if (...) else	logical choice
iff	logical equivalence
forall	universal quantification

Predicate Logic:

Operator	Description
&&	logical and
||	logical or
!	logical not
if (...)	logical implication
if (...) else	logical choice
iff	logical equivalence
forall	universal quantification
exists	existential quantification

Relational:

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

Arithmetic:

Operator	Description
+	addition
-	subtraction
/	division
*	multiplication

Logic Extensions:

Operator	Description

Logic Extensions:

Operator	Description
x'	value of x after method execution

Logic Extensions:

Operator	Description
x’	value of *x* after method execution
return	return value of method, as a variable

Logic Extensions Example:

```
abstract class X {
  int i;    // instance variable

  /** Increment i and return its value. */
  abstract int incrAndReturnI();
}
```

Logic Extensions Example:

```
abstract class X {
  int i;    // instance variable

  /** Increment i and return its value. */
  abstract int incrAndReturnI();
}
```

How do we specify this precisely?

Logic Extensions Example:

```
abstract class X {
  int i;    // instance variable

  /** Increment i and return its value.
    post: i’ == i+1 && return == i’;
   */
  abstract int incrAndReturnI();
}
```

Logic Extensions Example:

Wait a minute, that was some serious overkill.

Let’s just implement the silly little thing.

Logic Extensions Example:

```
class X {
  int i;    // instance variable

  /** Increment i and return its value. */
  int incrAndReturnI() {
    return i++;
  }
}
```

Logic Extensions Example:

Wait a minute, is that right?

Collections, Lists, Strings:

Collections, Lists, Strings:

Operator	Description

Collections, Lists, Strings:

Operator	Description
.size()	number of collection elements

Collections, Lists, Strings:

Operator	Description
.size()	number of collection elements
.contains(Object o)	collection membership

Collections, Lists, Strings:

Operator	Description
.size()	number of collection elements
.contains(Object o)	collection membership
.get(int i)	get ith list element

Collections, Lists, Strings:

Operator	Description
.size()	number of collection elements
.contains(Object o)	collection membership
.get(int i)	get ith list element
.length(String s)	length of s

V. “Programming” with predicates

V. “Programming” with predicates

A. Predicates are *non-procedural*.

V. **“Programming” with predicates**

A. Predicates are *non-procedural*.

B. Rules for this style of "programming":

V. “Programming” with predicates

A. Predicates are *non-procedural*.

B. Rules for this style of "programming":

1. Define what methods do, not how they work.

V. “Programming” with predicates

A. Predicates are *non-procedural*.

B. Rules for this style of "programming":

1. Define what methods do, not how they work.

2. Method code is only two boolean expressions -- ***nothing more.***

"Programming", cont’d

3. The closest thing to control constructs are quantifiers `forall` and `exists`.

"Programming", cont'd

3. The closest thing to control constructs are quantifiers `forall` and `exists`.

   a. But they're fundamentally different.

"Programming", cont'd

3. The closest thing to control constructs are quantifiers `forall` and `exists`.

   a. But they're fundamentally different.

   b. They're only boolean values
      -- nothing "happens".

"Programming", cont’d

4. Not executed like a statement.

"Programming", cont’d

4. Not executed like a statement.

   a. Time does not pass.

"Programming", cont’d

4. Not executed like a statement.

   a. Time does not pass.

   b. Conditions are mathematical fact, instantaneously true or false.

"Programming", cont’d

4. Not executed like a statement.

   a. Time does not pass.

   b. Conditions are mathematical fact, instantaneously true or false.

   c. I.e., `forall` is not a for-loop.

"Programming", cont’d

C. It may be necessary to specify order of ops.

"Programming", cont’d

C. It may be necessary to specify order of ops.

1. But not with conventional programming.

"Programming", cont’d

C. It may be necessary to specify order of ops.

1. But not with conventional programming.

2. We’ll specify ordering non-procedurally, using pre/post dependencies.

"Programming", cont’d

a. If method *B* must follow *A*, write pre- and postconditions accordingly.

"Programming", cont'd

a. If method *B* must follow *A*, write pre- and postconditions accordingly.

b. Specifically, *A*'s postcond specifies a unique condition that *B*'s precond requires.

VI. An initial formal spec example

VI. An initial formal spec example

A. Calendar Tool database ops.

VI. An initial formal spec example

A. Calendar Tool database ops.

B. For user and group databases.

VI. An initial formal spec example

A. Calendar Tool database ops.

B. For user and group databases.

C. Reasonably straightforward specs.

VI. An initial formal spec example

A. Calendar Tool database ops.

B. For user and group databases.

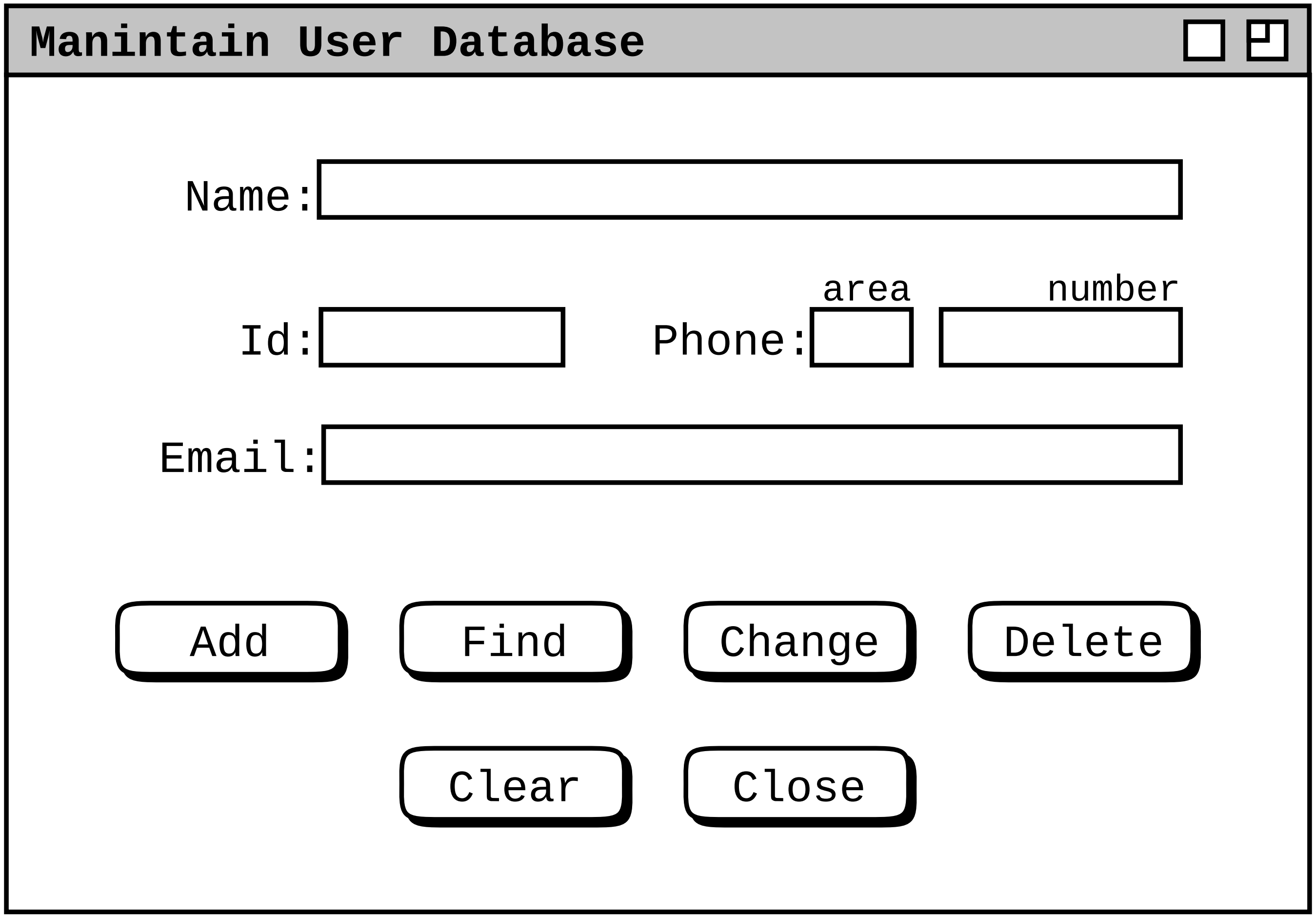
C. Reasonably straightforward specs.

D. Next week more involved.

VII. Synopsis of requirements.

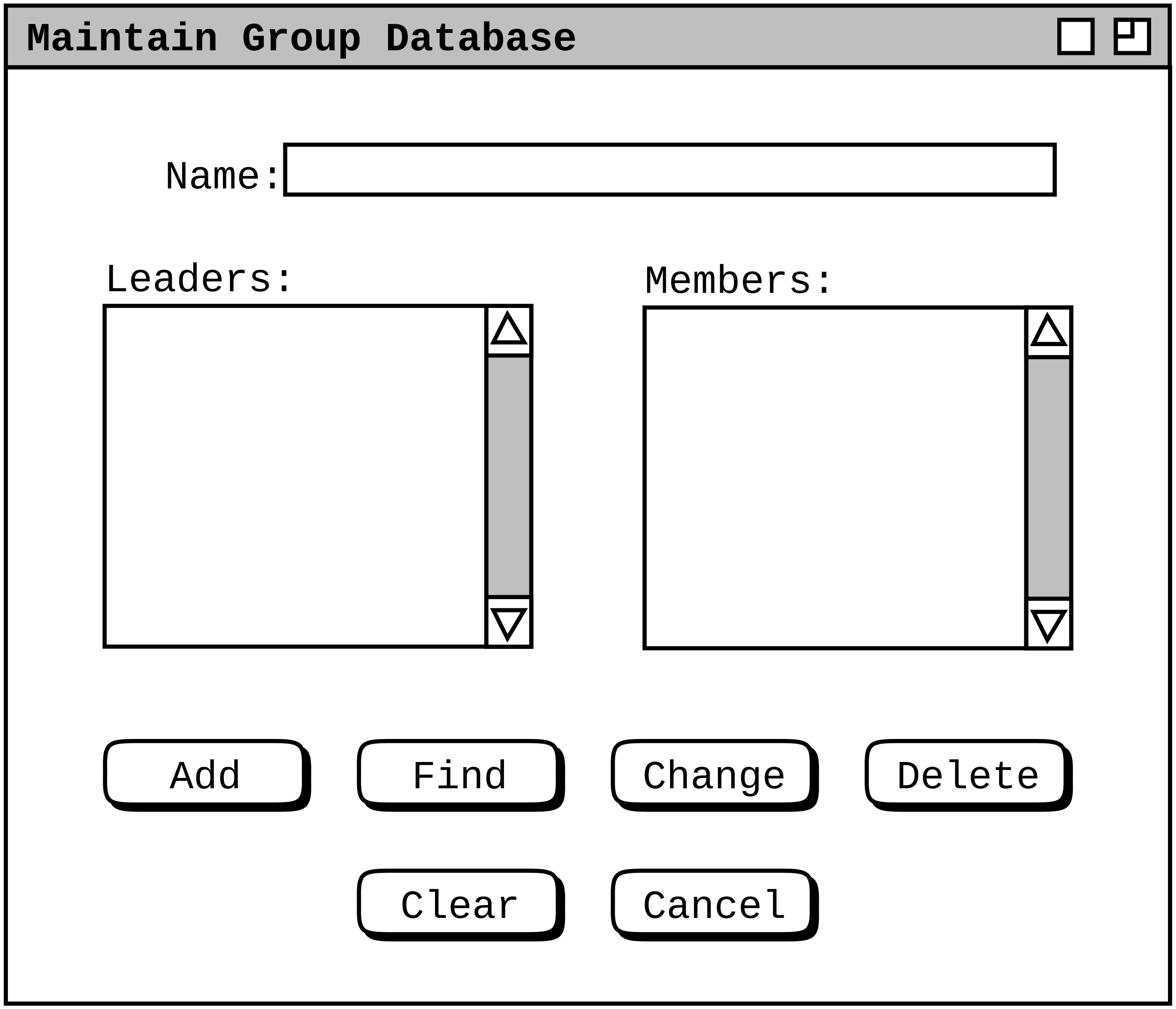
VII. Synopsis of requirements.

A. User selects `Admin->Users ...`


Manintain User Database
Name:
area
number
Id:
Phone:
Email:
Add
Find
Change
Delete
Clear
Close

Synopsis of requirements, cont’d

B. User selects `Admin>Groups ....`

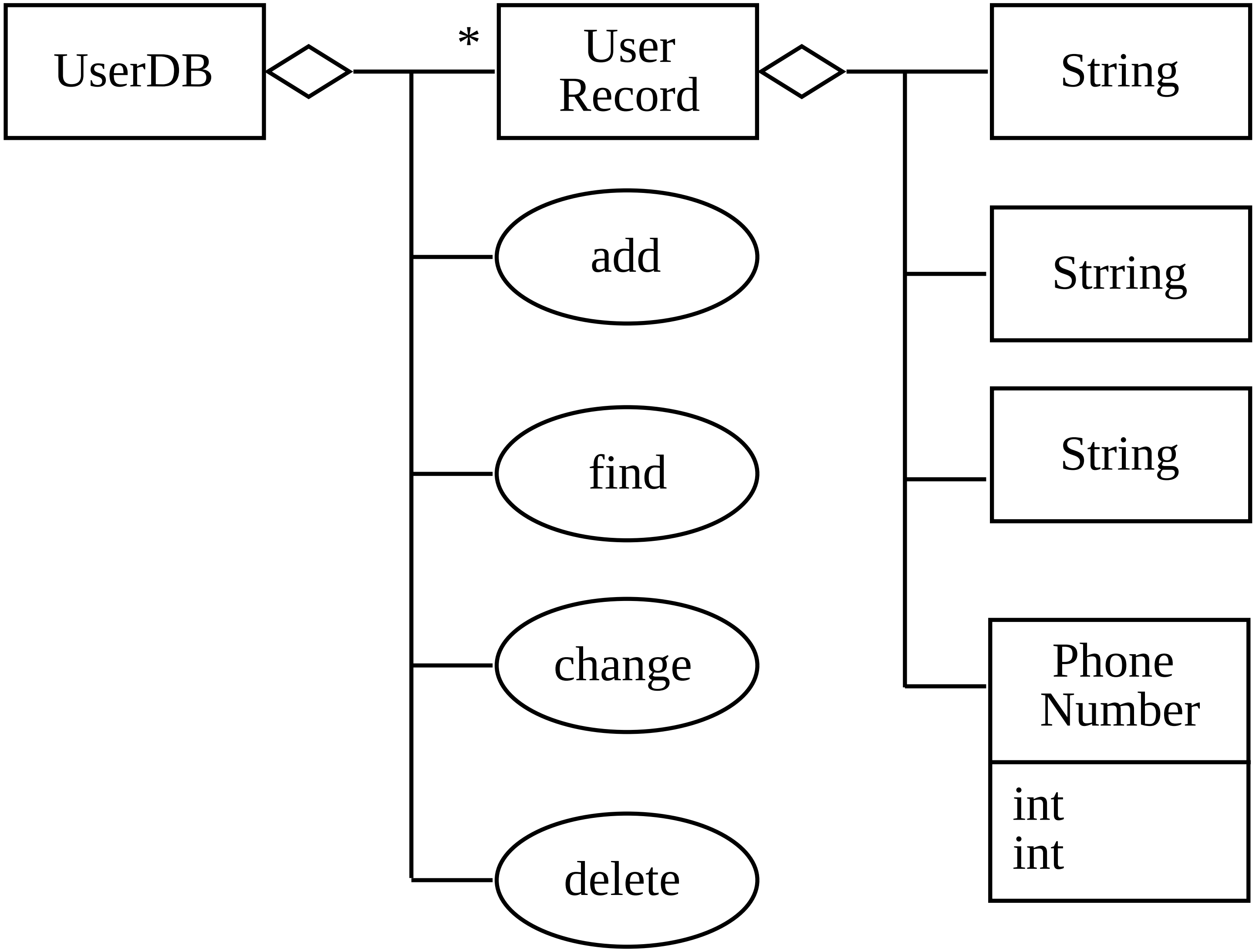

Maintain Group Database
Name:
Leaders:
Members:
Add
Find
Change
Delete
Clear
Cancel

VIII. Basic defs for user db objects and operations

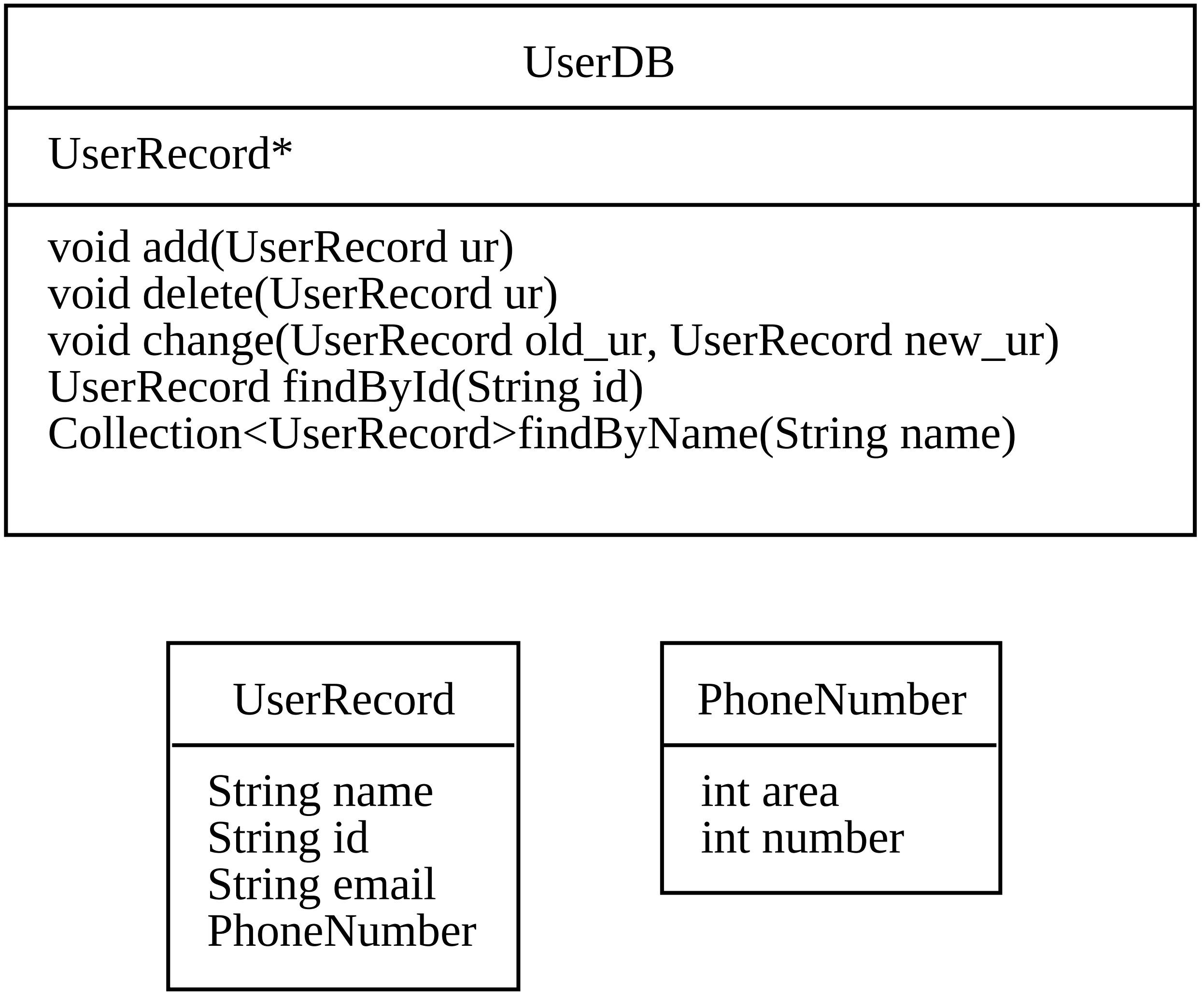
VIII. Basic defs for user db objects and operations

A. Here they are ...

B. UML diagram, one part


UserDB
*
User Record
add
find
change
delete
String
Strring
String
Phone Number
int
int

UML diagram, three part



UserDB

UserRecord*

void add(UserRecord ur)
void delete(UserRecord ur)
void change(UserRecord old_ur, UserRecord new_ur)
UserRecord findById(String id)
Collection<UserRecord>findByName(String name)

UserRecord

String name
String id
String email
PhoneNumber

PhoneNumber

int area
int number

Basic objs and ops, cont’d

C. Derived per Notes Week 4.

Basic objs and ops, cont’d

C. Derived per Notes Week 4.

D. Op signatures are representative.

Basic objs and ops, cont’d

1. `UserDB.add` is ***constructive***

```
class ACollection {

    Collection<AnElement> data;

    void constructiveOp(AnElement);

}
```

Basic objs and ops, cont’d

2. `UserDB.findBy...` are ***selective***.

```
class ACollection {

    AnElement selectiveOp(
        UniqueElementSelector);

    Collection<AnElement> selectiveOp(
        NonUniqueElementSelector);
}
```

Basic objs and ops, cont’d

3. `UserDB.delete` is ***destructive***

```
void destructiveOp(AnElement);
```

/* same signature as constructive */

Basic objs and ops, cont’d

4. `UserDB.change` is ***modifying***

```
class ACollection {

    void modifyingOp(
        AnElement oldElement,
        AnElement newElement);

}
```

IX. Initial formal definition of `UserDB.add`.

IX. Initial formal definition of `UserDB.add`.

A. Start in English.

IX. Initial formal definition of `UserDB.add`.

A. Start in English.

B. Refine logic.

IX. Initial formal definition of `UserDB.add`.

A. Start in English.

B. Refine logic.

C. So,

Formal `UserDB.add`, cont’d

```
abstract class UserDB {
    Collection<UserRecord> data;
    abstract void add(UserRecord ur);
```

Formal `UserDB.add`, cont’d

```
pre:
  //
  // 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.
  //
```

Formal `UserDB.add`, cont'd

```
post:
  //
  // The given user record is in
  // the output data.
  //
```

D. Formalizing the logic.

D. Formalizing the logic.

1. Postcond comment specifies fundamental property of an additive op --

E. **Formalizing the logic.**

1. Postcond comment specifies fundamental property of an additive op --

   the added element is in the output.

F. Formalizing the logic.

1. Postcond comment specifies fundamental property of an additive op --

   the added element is in the output.

2. So here it is,

Formalized Spest logic, cont’d

```
/*

  pre:
    // Coming soon

  post:
    // The given user record is in
    // the output data.
    data’.contains(ur);

 */
 abstract void add(UserRecord ur);
```

Formalized Spest logic, dissected

```
/*
```

Spest is in comments

Formalized Spest logic, dissected

```
/*
```

Spest is in comments

```
  pre:
```

Spest keyword

Formalized Spest logic, dissected

`/*`	***Spest is in comments***
`  pre:`	***Spest keyword***
`    // Coming soon;`	***Standard Java Comment***

Formalized Spest logic, dissected

`/*`	***Spest is in comments***
`  pre:`	***Spest keyword***
`    // Coming soon;`	***Standard Java Comment***
`  post:`	***Spest keyword***

Formalized Spest logic, dissected

```
/*

  pre:
    // Coming soon;

  post:
    // The given user record is in
    // the output data.
```

Spest is in comments

Spest keyword

Standard Java Comment

Spest keyword

Standard Java comment

Formalized Spest logic, dissected

```
/*

  pre:
    // Coming soon;

  post:
    // The given user record is in
    // the output data.
    data’.contains(ur);
```

Spest is in comments

Spest keyword

Standard Java Comment

Spest keyword

Standard Java comment

Java boolean expression,
using "prime" notation

Formalized Spest logic, dissected

```
/*

  pre:
    // Coming soon;

  post:
    // The given user record is in
    // the output data.
    data’.contains(ur);

 */
```

Spest is in comments

Spest keyword

Standard Java Comment

Spest keyword

Standard Java comment

Java boolean expression using "prime" notation

End Spest comment

Formalized Spest logic, dissected

```
/*

  pre:
    // Coming soon;

  post:
    // The given user record is in
    // the output data.
    data’.contains(ur);

 */
 abstract void add(UserRecord ur);
```

Spest is in comments

Spest keyword

Standard Java Comment

Spest keyword

Standard Java comment

Java boolean expression using "prime" notation

End Spest comment

Method signature

Formalized Spest logic, cont’d

3. `data’.contains(ur)` is all there is.

Formalized Spest logic, cont'd

3. `data'.contains(ur)` is all there is.

   a. Only non-standard syntax is the apostrophe.

Formalized Spest logic, cont’d

3. `data’.contains(ur)` is all there is.

   a. Only non-standard syntax is the apostrophe.

   b. `contains` is `Collection` method.

Formalized Spest logic, cont’d

3. `data’.contains(ur)` is all there is.

   a. Only non-standard syntax is the apostrophe.

   b. `contains` is `Collection` method.

   c. Its operand is element in collection.

Formalized Spest logic, cont’d

3. `data’.contains(ur)` is all there is.

   a. Only non-standard syntax is the apostrophe.

   b. `contains` is `Collection` method.

   c. Its operand is element in collection.

   d. I.e., a `UserRecord`.

Formalized Spest logic, cont’d

G. `UserDB.add` still has no formal precond.

Formalized Spest logic, cont'd

G. `UserDB.add` still has no formal precond.

1. Empty precond equivalent to *true*.

Formalized Spest logic, cont’d

G. `UserDB.add` still has no formal precond.

1. Empty precond equivalent to *true*.

2. In many cases this is fine.

Formalized Spest logic, cont’d

G. `UserDB.add` still has no formal precond.

1. Empty precond equivalent to *true*.
2. In many cases this is fine.
3. Won’t do in this case.

Formalized Spest logic, cont'd

G. `UserDB.add` still has no formal precond.

1. Empty precond equivalent to *true*.
2. In many cases this is fine.
3. Won't do in this case.
4. We'll do formal precond soon.

X. Refining `UserDB.add` postcond.

X. Refining `UserDB.add` postcond.

A. A fundamental question is --
are conditions *strong enough*?

X. Refining `UserDB.add` postcond.

A. A fundamental question is --
are conditions *strong enough*?

1. Adding logic clauses strengthens.

X. Refining `UserDB.add` postcond.

A. A fundamental question is --
are conditions *strong enough*?

1. Adding logic clauses strengthens.

2. E.g., true precond is weaker than precond "no `UserRecord` of same Id in db".

Refining `UserDB.add` postcond, cont'd

B. Two aims in strengthening conditions:

Refining `UserDB.add` postcond, cont’d

B. Two aims in strengthening conditions:

1. Ensure user requirements are met (Maxim 1).

Refining `UserDB.add` postcond, cont'd

B. Two aims in strengthening conditions:

1. Ensure user requirements are met (Maxim 1).

2. Ensure implementation works (Maxim 2).

Refining `UserDB.add` postcond, cont'd

C. Accomplishing these requires

Refining `UserDB.add` postcond, cont’d

C. Accomplishing these requires

1. copious user consultation

Refining `UserDB.add` postcond, cont'd

C. Accomplishing these requires

1. copious user consultation

2. an analyst who understands potential implementation problems

Refining `UserDB.add` postcond, cont'd

D. For additive methods, potential implementation error is ***spurious addition or deletion***.

Refining `UserDB.add` postcond, cont'd

D. For additive methods, potential implementation error is ***spurious addition or deletion***.

E. To avoid, strengthen as follows:

Refining `UserDB.add` postcond, cont’d

```
post:

  // The given user record is in
  // the output data.
  data’.contains(ur)

      &&
```

Refining `UserDB.add` postcond, cont’d

```
//
// For any other user record
// that’s not the input record,
// it’s in the output data if and
// only if it’s in the input data
//
```

Refining `UserDB.add` postcond, cont’d

```
//
// For any other user record
// that’s not the input record,
// it’s in the output data if and
// only if it’s in the input data
//
forall (UserRecord ur_other ;
        !ur_other.equals(ur) ;
    if (data.contains(ur_other))
        data’.contains(ur_other)
    else
        !data’.contains(ur_other))
```

Refining `UserDB.add` postcond, cont’d

Let’s dissect this logic.

Refining `UserDB.add` postcond, cont’d

```
//
// For any other user record
// that’s not the input record,
// it’s in the output data if and
// only if it’s in the input data
//
forall (UserRecord ur_other ;
```

quantifier

Refining `UserDB.add` postcond, cont’d

```
//
// For any other user record
// that’s not the input record,
// it’s in the output data if and
// only if it’s in the input data
//
forall (UserRecord ur_other ;   quantifier
        !ur_other.equals(ur) ;  constraint
```

Refining `UserDB.add` postcond, cont’d

```
//
// For any other user record
// that’s not the input record,
// it’s in the output data if and
// only if it’s in the input data
//
forall (UserRecord ur_other ;   quantifier
        !ur_other.equals(ur) ;  constraint
                                predicate:
    if (data.contains(ur_other))
        data’.contains(ur_other)
    else
        !data’.contains(ur_other))
```

Refining `UserDB.add` postcond, cont’d

F. Introduces `forall`.

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

2. General form:

`forall` (*T x* ; *constraint* ; *predicate*)

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

2. General form:

   `forall` (*T x* ; *constraint* ; *predicate*)

   Read as:

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

2. General form:

`forall` (*T x* `;` *constraint* `;` *predicate*)

Read as:

"For all values *x* of type *T*,

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

2. General form:

`forall` (*T x* ; *constraint* ; *predicate*)

Read as:

"For all values *x* of type *T*,

such that *constraint* holds,

Refining `UserDB.add` postcond, cont'd

F. Introduces `forall`.

1. Same meaning as standard math logic.

2. General form:

`forall` (*T x* ; *constraint* ; *predicate*)

Read as:

"For all values *x* of type *T*,
such that *constraint* holds,
predicate is true."

Refining `UserDB.add` postcond, cont'd

3. *Constraint* expression is optional.

Refining `UserDB.add` postcond, cont’d

3. *Constraint* expression is optional.

4. The quantified variable *x* must appear in *constraint* (if present) and *predicate*.

Refining `UserDB.add` postcond, cont’d

G. There’s an easier way:

1. E.g.,

Refining `UserDB.add` postcond, cont'd

```
post:
  //
  // A user record is in the output data
  // if and only if it is the new record
  // to be added or is in the input data.
  //
```

Refining `UserDB.add` postcond, cont’d

```
post:
  //
  // A user record is in the output data
  // if and only if it is the new record
  // to be added or is in the input data.
  //
  forall (UserRecord ur_other ;
      data’.contains(ur_other) iff
         ur_other.equals(ur) ||
            data.contains(ur_other))
```

Refining `UserDB.add` postcond, cont’d

2. Such logic simplification is beneficial when it helps clarify.

Refining `UserDB.add` postcond, cont’d

2. Such logic simplification is beneficial when it helps clarify.

3. Simplification is not necessary as long as logic is clear and accurate.

Refining `UserDB.add` postcond, cont'd

H. What about using `Collection.add`?

```
/*
 post:

  data.add(ur);

 */
abstract void add(UserRecord ur);
```

Refining `UserDB.add` postcond, cont’d

1. Does `Collection.add` do what we want?

Refining `UserDB.add` postcond, cont’d

1. Does `Collection.add` do what we want?

2. We don’t know, since it’s abstract.

Refining `UserDB.add` postcond, cont'd

1. Does `Collection.add` do what we want?

2. We don't know, since it's abstract.

3. So, we can't rely on it here.

XI. Refining other `UserDB` ops.

A. Other ops comparable to `UserDB.add`.

B. Here’s `findByName`:

Refining other ops, cont’d

```
import java.util.Collection;

abstract class UserDB {

  Collection<UserRecord> data;
```

Refining other ops, cont’d

```
import java.util.Collection;

abstract class UserDB {

  Collection<UserRecord> data;

  /**
   * Find a user or users by real-world
   * name.  If more than one is found,
   * the output list is sorted by id.
   */
```

```
 post:
   //
   // A record is in the output list iff
   // it’s in the input and it’s name
   // is what’s being searched for.
   //

*/

Collection<UserRecord>
      findByName(String name);
 post:
   //
   // A record is in the output list iff
   // it’s in the input and it’s name
   // is what’s being searched for.
   //
   forall (UserRecord ur ;
     return.contains(ur) iff
       data.contains(ur) &&
         ur.name.equals(name));

*/

Collection<UserRecord>
      findByName(String name);
```

XII. Using quantifiers

XII. Using quantifiers

A. Universal and existential quantification state multiple conditions in a single expression.

XII. Using quantifiers

A. Universal and existential quantification state multiple conditions in a single expression.

1. Universal quantification `forall` is true if *all* cases are true.

XII. Using quantifiers

A. Universal and existential quantification state multiple conditions in a single expression.

1. Universal quantification `forall` is true if *all* cases are true.

2. Existential quantification `exists` is true if *at least one* case is true.

Quantifiers, cont’d

3. Think of `forall` and `exists` as repeated `and` and `or`, respectively

Quantifiers, cont’d

4. There’s a generalized DeMorgan’s law

```
forall (T x ; p) iff
    !exists (T x ; !p)

!forall (T x;  !p) iff
    exists (T x ; p)
```

Quantifiers, cont’d

B. For us, using quantifiers has two objectives:

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

```
forall (UserRecord ur ;
    requirement-predicate)
```

Quantifiers, cont’d

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

   ```
   exists (UserRecord ur ;
     requirement-predicate
   ```

Quantifiers, cont’d

C. Specialized forms provide further focus.

1. Requirement about values in a collection, e.g.,

```
forall (UserRecord ur ;
    data.contains(ur);
      requirement-predicate)

exists (UserRecord ur ;
    data.contains(ur);
      requirement-predicate)
```

Quantifiers, cont’d

2. Requirement about values, with some further restrictions, e.g.,

```
forall (int i ; i > 0 ;
  requirement-predicate)

exists (int i ; i > 0 ;
  requirement-predicate)
```

Quantifiers, cont’d

D. Specific focuses help narrow down when and how to use quantifiers.

XIII. Formally spec’ing user-level requirements

XIII. Formally spec’ing user-level requirements

A. So far we’ve done basic DB requirements.

XIII. Formally spec’ing user-level requirements

A. So far we’ve done basic DB requirements.

B. We’ve focused on the second maxim
-- *not trusting the programmer.*

XIII. Formally spec’ing user-level requirements

A. So far we’ve done basic DB requirements.

B. We’ve focused on the second maxim
-- *not trusting the programmer.*

C. Now for first maxim
-- *nothing is obvious*.

User-level requirements, cont’d

D. Here are three "obvious" requirements:

User-level requirements, cont’d

D. Here are three "obvious" requirements:

1. No duplicate entries allowed in `UserDB`.

User-level requirements, cont’d

D. Here are three "obvious" requirements:

1. No duplicate entries allowed in `UserDB`.

2. Input records are checked for validity.

User-level requirements, cont’d

D. Here are three "obvious" requirements:

1. No duplicate entries allowed in `UserDB`.

2. Input records are checked for validity.

3. If `UserDB.find` outputs more than one record, output is sorted appropriately.

XIV. No duplicates requirement.

XIV. No duplicates requirement.

A. It’s not really obvious what "duplicate" means.

XIV. No duplicates requirement.

A. It’s not really obvious what "duplicate" means.

B. A number of plausible interpretations.

XIV. No duplicates requirement.

A. It’s not really obvious what "duplicate" means.

B. A number of plausible interpretations.

C. Here are three possibilities:

No duplicates, cont’d

1. No records in `UserDB` have same values for all components.

No duplicates, cont’d

1. No records in `UserDB` have same values for all components.

2. No records in `UserDB` have same name.

No duplicates, cont’d

1. No records in `UserDB` have same values for all components.

2. No records in `UserDB` have same name.

3. No records in `UserDB` have same id.

No duplicates, cont’d

C. Which interpretation to choose is *not* for the programmer alone to decide.

No duplicates, cont’d

C. Which interpretation to choose is *not* for the programmer alone to decide.

1. Decided by analyst in consult with end users.

No duplicates, cont'd

C. Which interpretation to choose is *not* for the programmer alone to decide.

1. Decided by analyst in consult with end users.

2. Even if we grant that most programmers are reasonably smart.

No duplicates, cont’d

D. Here, we have determined with customer that `Id` component is unique key.

No duplicates, cont'd

D. Here, we have determined with customer that `Id` component is unique key.

1. `UserRecord`s need only differ by Id.

No duplicates, cont'd

D. Here, we have determined with customer that `Id` component is unique key.

1. `UserRecord`s need only differ by Id.

2. Multiple with the same name is OK.

No duplicates, cont’d

E. Basic strategy is precond on `UserDB.add`.

No duplicates, cont’d

E. Basic strategy is precond on `UserDB.add`.

F. Here is the refined spec:

No duplicates, cont’d

```
/*
  pre:
    //
    // There is no user record in the
    // input data with the same id as
    // the record to be added.
    //
```

No duplicates, cont’d

```
/*
  pre:
    //
    // There is no user record in the
    // input data with the same id as
    // the record to be added.
    //
    !exists (UserRecord ur_existing ;
        data.contains(ur_existing) ;
            ur_existing.id.equals(ur.id));
*/
```

No duplicates, cont’d

```
/*
  pre:
    //
    // There is no user record in the
    // input data with the same id as
    // the record to be added.
    //
    !exists (UserRecord ur_existing ;
        data.contains(ur_existing) ;
            ur_existing.id.equals(ur.id));
*/
void add(UserRecord ur);
```

XV. Input value checking

XV. Input value checking

A. Described in scenarios as follows:

XV. Input value checking

A. Described in scenarios as follows:

1. Id unique, <= 8 chars

XV. Input value checking

A. Described in scenarios as follows:

1. Id unique, <= 8 chars

2. Email address free-form string

XV. Input value checking

A. Described in scenarios as follows:

1. Id unique, <= 8 chars

2. Email address free-form string

3. Area code 3 digits, number 7 digits

Input value checking, cont’d

B. Defined formally as follows

```
/*
  pre:
    //
    // No dups condition from above
    //
```

Input value checking, cont'd

B. Defined formally as follows

```
/*
  pre:
    //
    // No dups condition from above
    //
    !exists ( ... )

        &&
```

Input value checking, cont’d

```
//
// The id of the given user record is
// not empty and 8 characters or less.
//
```

Input value checking, cont’d

```
//
// The id of the given user record is
// not empty and 8 characters or less.
//
(ur.id != null) && (ur.id.length() > 0) &&
        (ur.id.length() <= 8)

    &&
```

Input value checking, cont’d

```
    //
    // The email address is not empty.
    //
```

Input value checking, cont’d

```
//
// The email address is not empty.
//
(ur.email != null) && (ur.email.length() > 0)
```

Input value checking, cont’d

```
  //
  // If phone area code and number are present,
  // they must be 3 digits and 7 digits respectively.
  //
```

Input value checking, cont’d

```
//
// If phone area code and number are present,
// they must be 3 digits and 7 digits respectively.
//
if (ur.phone.area != 0)
 Integer.toString(ur.phone.area).length() == 3)
   &&
if (ur.phone.number != 0)
 Integer.toString(ur.phone.number).length() == 7)
```

XVI. Ordering of multi-record output lists

XVI. Ordering of multi-record output lists

A. `UserDB.findByName` returns a list.

XVI. Ordering of multi-record output lists

A. `UserDB.findByName` returns a list.

B. Initial requirements unclear about ordering.

XVI. Ordering of multi-record output lists

A. `UserDB.findByName` returns a list.

B. Initial requirements unclear about ordering.

C. Reasonable choice is to sort the list by Id.

Ordering output lists, cont’d

1. Scenario updated to reflect decision.

Ordering output lists, cont’d

1. Scenario updated to reflect decision.

2. Joint decision of user and analyst.

Ordering output lists, cont’d

1. Scenario updated to reflect decision.

2. Joint decision of user and analyst.

3. Illustrates benefit of iterative approach to requirements analysis and modeling.

Ordering output lists, cont’d

D. We must update method return type and strengthen postcondition:

Ordering output lists, cont’d

D. We must update method return type and strengthen postcondition:

```
abstract List<UserRecord>// from Collection to List
        findByName(String name);
```

Ordering output lists, cont’d

D. We must update method return type and strengthen postcondition:

```
abstract List<UserRecord>// from Collection to List
        findByName(String name);
    /*
      post:
```

Ordering output lists, cont’d

```
//
// The output list consists of all records
// of the given name in the input data.
```

Ordering output lists, cont'd

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)
```

Ordering output lists, cont’d

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)

forall (UserRecord ur ;  // for all user records
```

Ordering output lists, cont’d

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)

forall (UserRecord ur ;   // for all user records
  return.contains(ur) ;   //   in the output
```

Ordering output lists, cont’d

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)

forall (UserRecord ur ;  // for all user records
  return.contains(ur) ;  //   in the output
    data.contains(ur) && // it’s in the input and
```

Ordering output lists, cont’d

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)

forall (UserRecord ur ;  // for all user records
  return.contains(ur) ;  //   in output
    data.contains(ur) && // it’s in the input and
        ur.name.equals(name)) // has desired name
```

Ordering output lists, cont'd

```
//
// The output list consists of all records
// of the given name in the input data.
// (This is the initial postcond without sorting.)

forall (UserRecord ur ;  // for all user records
  return.contains(ur) ;  //   in output
    data.contains(ur) && // it's in the input and
        ur.name.equals(name)) // has desired name

      &&    // Now strengthen for sorting ...
```

Ordering output lists, cont’d

```
//
// The output list is sorted by id
// according to the semantics of
// java.lang.String.compareTo().
//
```

Ordering output lists, cont’d

```
//
// The output list is sorted by id
// according to the semantics of
// java.lang.String.compareTo().
//
forall (int i ;   // quantify over ints
```

Ordering output lists, cont’d

```
//
// The output list is sorted by id
// according to the semantics of
// java.lang.String.compareTo().
//
forall (int i ;    // quantify over ints
    (i >= 0) &&    // constrain to list range
      (i < return.size() - 1) ;
```

Ordering output lists, cont’d

```
//
// The output list is sorted by id
// according to the semantics of
// java.lang.String.compareTo().
//
forall (int i ;   // quantify over ints
    (i >= 0) &&   // constrain to list range
      (i < return.size() - 1) ;
                  // compare adjacent elements
        return.get(i).id.compareTo(
            return.get(i+1).id) < 0);
}
```

Ordering output lists, cont'd

E. English translation of sorting logic:

Ordering output lists, cont’d

E. English translation of sorting logic:

"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."

Ordering output lists, cont’d

E. English translation of sorting logic:

"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. Study this logic to be satisfied.

-- Additional Topics from Detailed Notes --

-- Additional Topics from Detailed Notes --

XVII. **Unbounded quantification**

-- Additional Topics from Detailed Notes --

XVII. **Unbounded quantification**

XVIII. **Using auxiliary functions**

-- Additional Topics from Detailed Notes --

XVII. **Unbounded quantification**

XVIII. **Using auxiliary functions**

XIX. **Specs for the GroupDB**