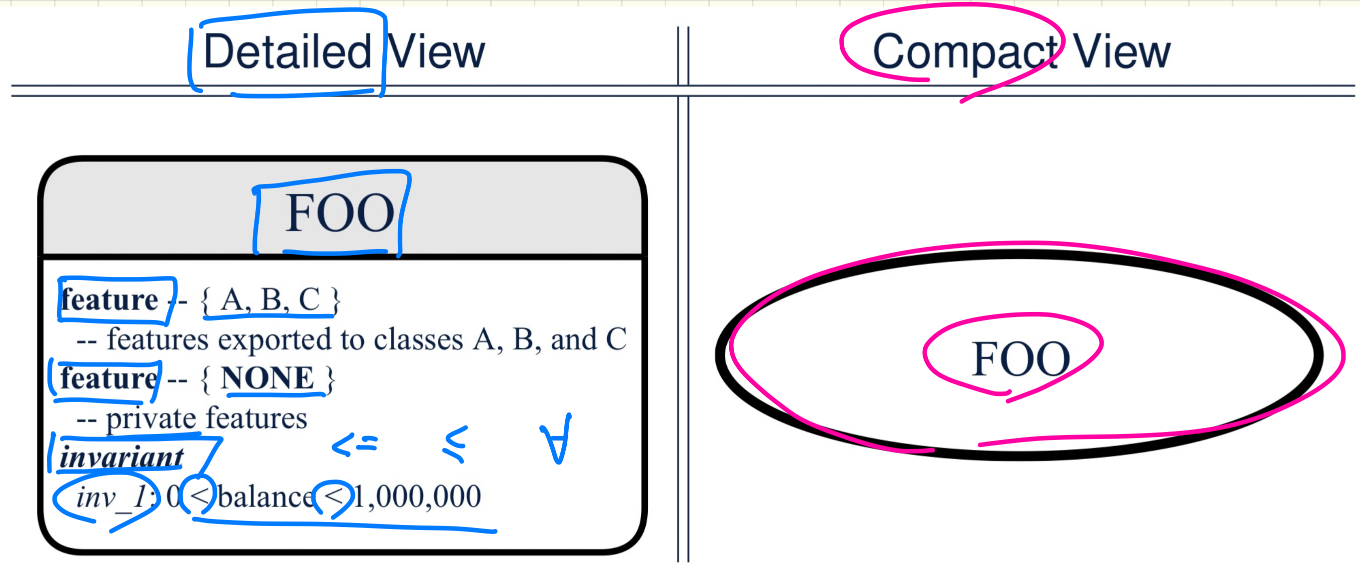
Lecture 4

Part 1

Architectural Design Diagrams

Classes: Detailed vs. Compact



Contracts: Mathematical vs Programming

```
ARRAYED_CONTAINER+

feature -- Queries
  count+: INTEGER
    -- Number of items stored in the container

feature -- Commands
  assign_at+ (i: INTEGER; s: STRING)
    -- Change the value at position 'i' to 's'.
  require
    valid_index: 1 ≤ i ≤ count
  ensure
    size_unchanged: imp.count = (old imp.twin).count
    item_assigned: imp[i] ~ s
    others_unchanged: ∀j : 1 ≤ j ≤ imp.count : j ≠ i ⇒ imp[j] ~ (old imp.twin) [j]

feature -- { NONE }
  imp+: ARRAY[STRING]
    -- Implementation of an arrayed-container

invariant
  consistency: imp.count = count
```

effective

public

public

private

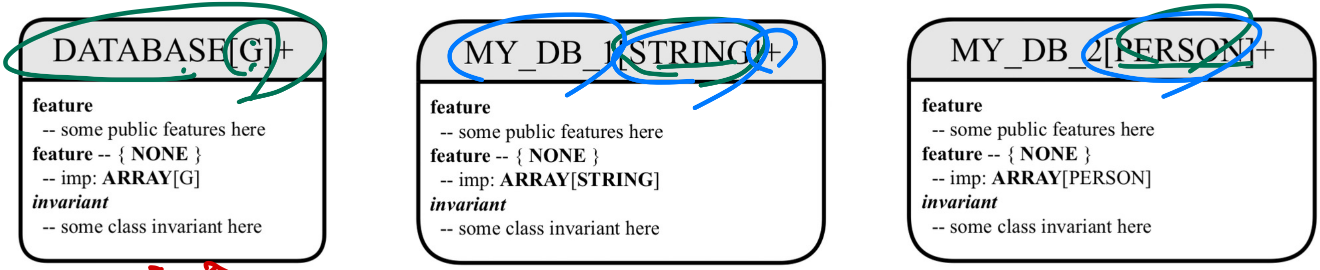
\+ → effective

* → deferred

Generic Classes

- Type parameter(s) of a class may or may not be *instantiated*:

- If necessary, present a generic class in the detailed form:

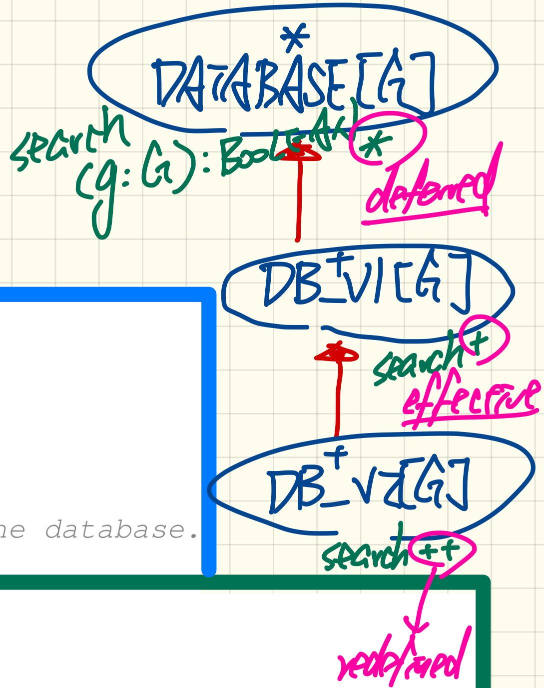


Programming Classes: Deferred vs. Effective

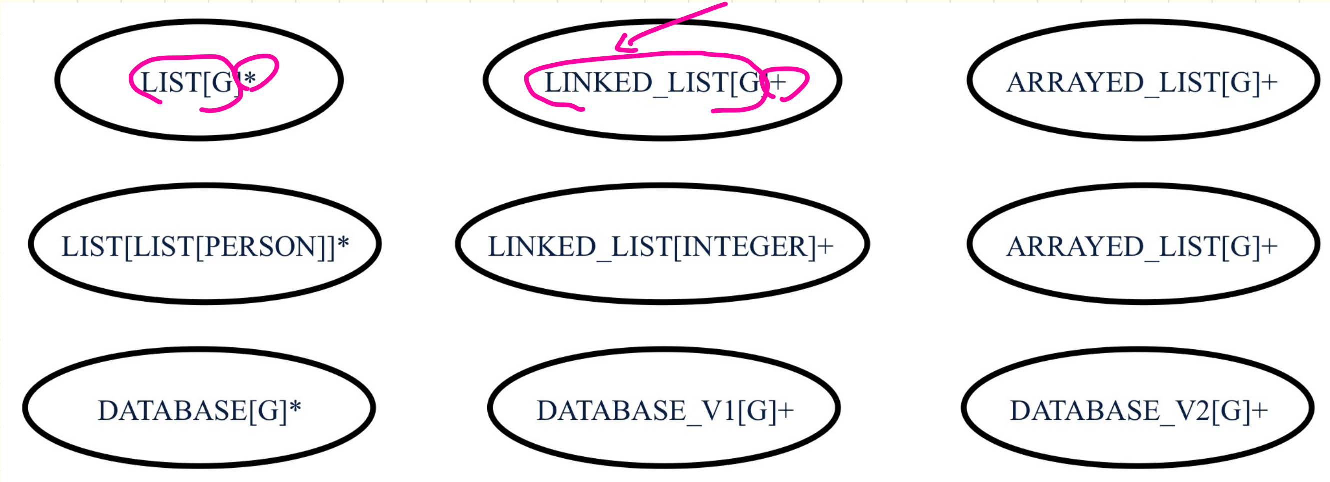
```
deferred class
  DATABASE[G]
feature -- Queries
  search (g: G): BOOLEAN
    -- Does item 'g' exist in database?
  deferred end
end
```

```
class
  DATABASE_V1[G]
inherit
  DATABASE[G]
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a linear search on the database.
  do end
end
```

```
class
  DATABASE_V2[G]
inherit
  DATABASE_V1[G]
    redefine search end
feature -- Queries
  search (g: G): BOOLEAN
    -- Perform a binary search on the database.
  do end
end
```



Presenting Deferred/Effective Classes in Compact Form



Presenting Deferred/Effective Classes in Detailed Form

DATABASE[G]*

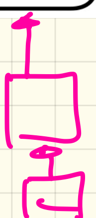
feature {NONE} -- Implementation
data: **ARRAY**[G]

feature -- Commands
add_item* (g: G)
-- Add new item `g` into database.
require
non_existing_item: $\neg$ exists (g)
ensure
size_incremented: count = **old** count + 1
item_added: exists (g)

feature -- Queries
count+: **INTEGER**
-- Number of items stored in database
ensure
correct_result: **Result** = data.count

exists* (g: G): **BOOLEAN**
-- Does item `g` exist in database?
ensure
correct_result: **Result** = $(\exists i : 1 \leq i \leq count : data[i] \sim g)$

not ×



DATABASE_V1[G]+

feature {NONE} -- Implementation
data: **ARRAY**[G]

feature -- Commands
add_item+ (g: G)
-- Append new item `g` into end of `data`.

feature -- Queries
count+: **INTEGER**
-- Number of items stored in database

exists+ (g: G): **BOOLEAN**
-- Perform a linear search on `data` array.

DATABASE_V2[G]+

feature {NONE} -- Implementation
data: **ARRAY**[G]

feature -- Commands
add_item++ (g: G)
-- Insert new item `g` into the right slot of `data`.

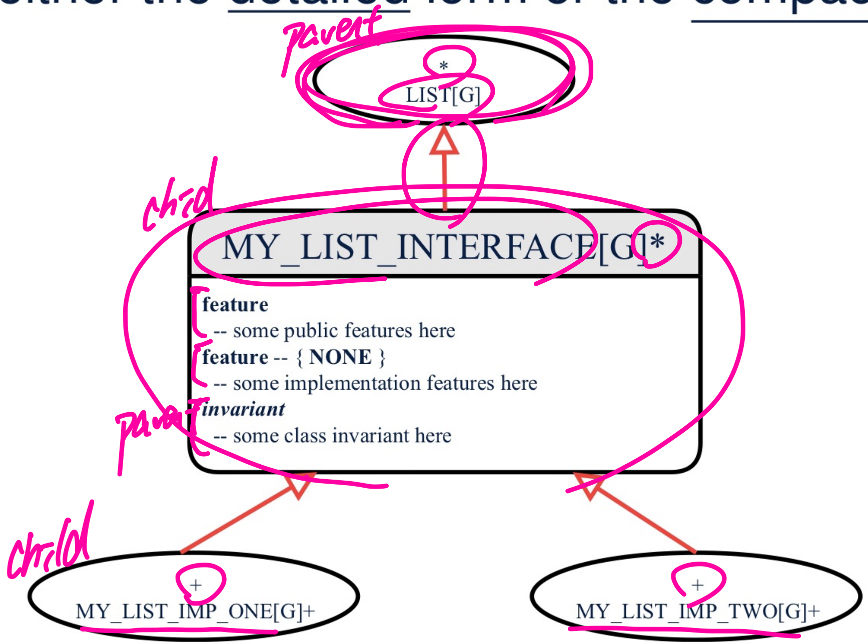
feature -- Queries
count+: **INTEGER**
-- Number of items stored in database

exists++ (g: G): **BOOLEAN**
-- Perform a binary search on `data` array.

invariant
sorted_data: $\forall i : 1 \leq i < count : data[i] < data[i+1]$

Inheritace (1)

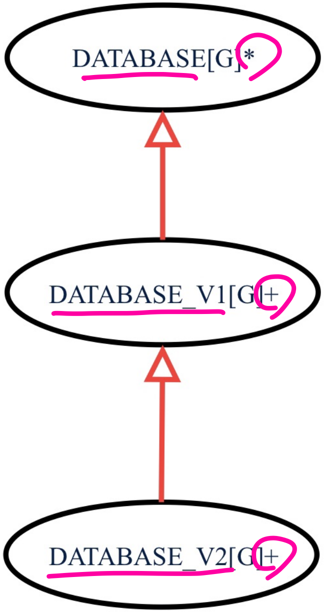
- You may choose to present each class in an inheritance hierarchy in either the detailed form or the compact form:

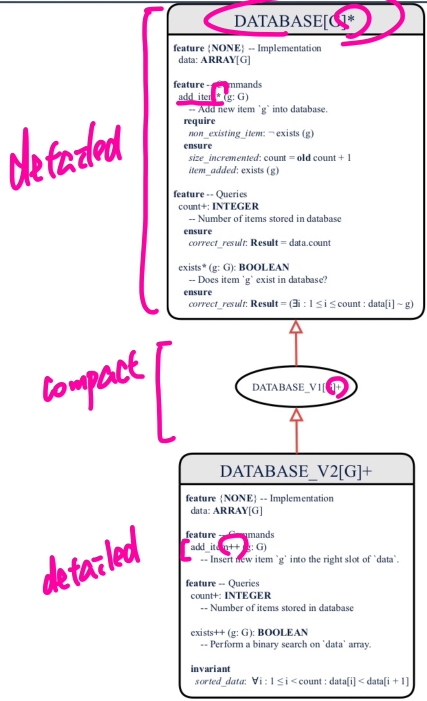


Inheritace (2)

More examples (emphasizing different aspects of `DATABASE`):

Inheritance Hierarchy	Features being (Re-)Implemented





Programming Client-Supplier Relation

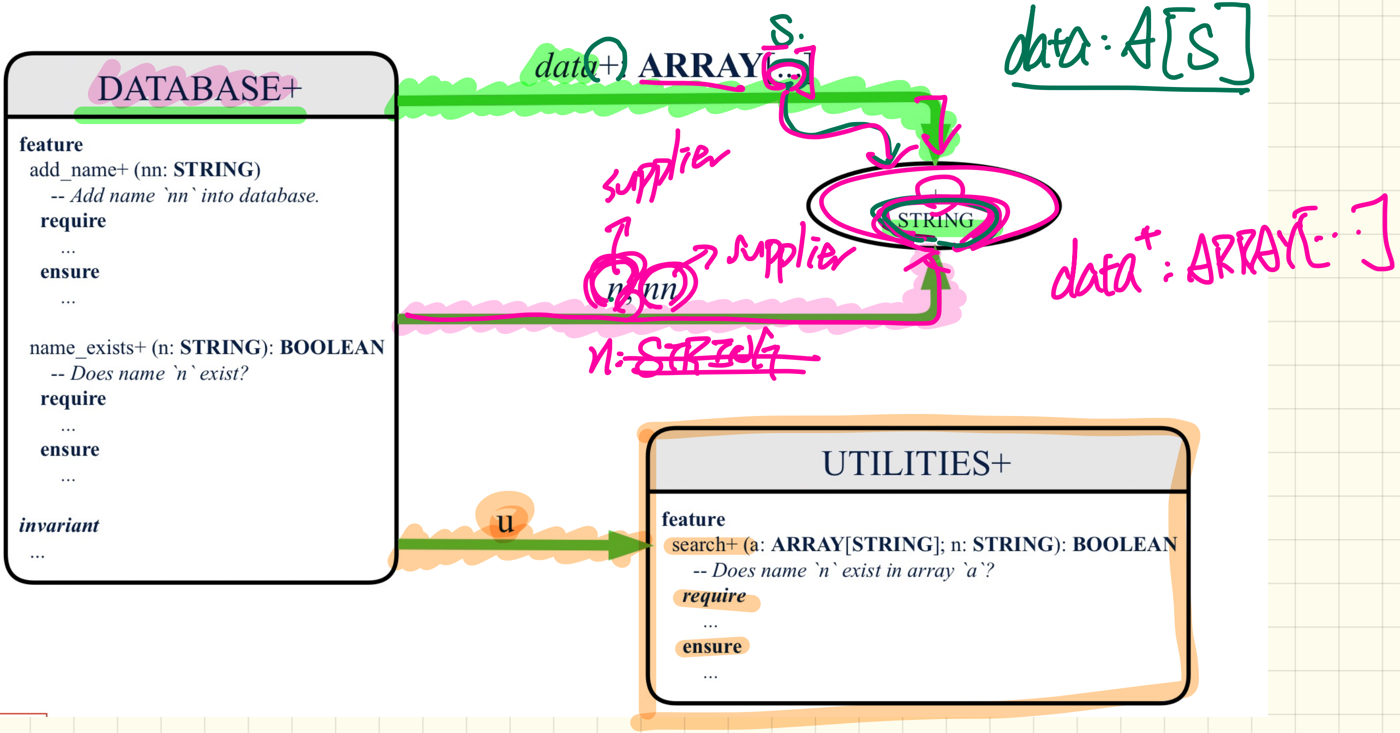
```
class DATABASE
feature {NONE} -- implementation
  data: ARRAY[STRING]
feature -- Commands
  add_name (nn: STRING)
      -- Add name 'nn' to database.
    require ... do ... ensure ... end

  name_exists (n: STRING): BOOLEAN
      -- Does name 'n' exist in database?
    require ...
    local
      u: UTILITIES
    do ... ensure ... end
invariant
  ...
end
```

→ query × [do / end]

```
class UTILITIES
feature -- Queries
  search (a: ARRAY[STRING]; n: STRING): BOOLEAN
      -- Does name 'n' exist in array 'a'?
    require ... do ... ensure ... end
end
```

Presenting CS Relation in Diagram: Approach 1

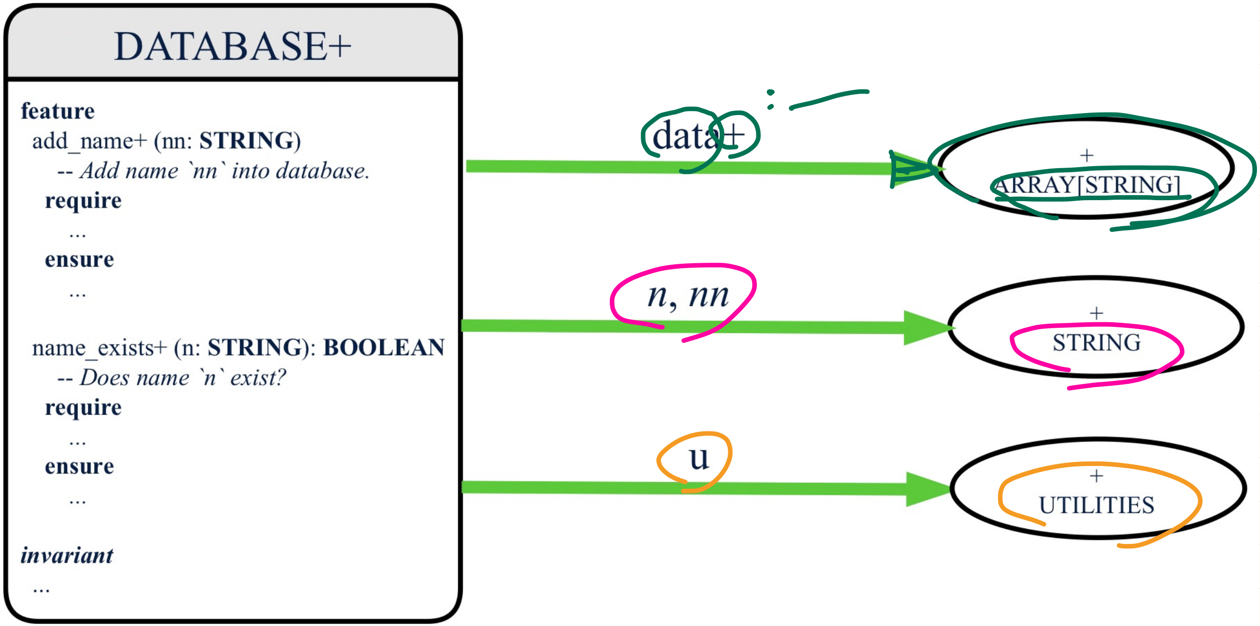
If `STRING` is to be emphasized, label is `data: ARRAY[...]`, where ... denotes the supplier class `STRING` being pointed to.



Presenting CS Relation in Diagram: Approach 2

If `ARRAY` is to be emphasized, label is `data`.

The supplier's name should be complete: `ARRAY[STRING]`



Programming Client-Supplier Relation

DESIGN ONE:

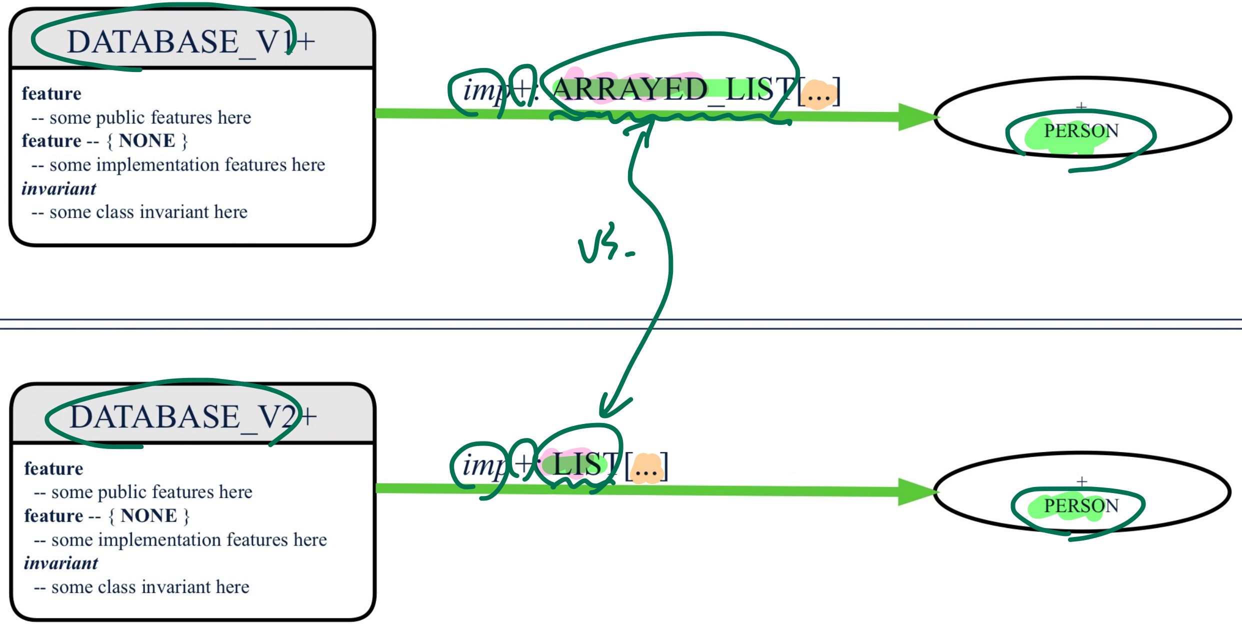
```
class DATABASE_V1
feature {NONE} -- implementation
  imp: ARRAYED_LIST[PERSON]
... -- more features and contracts
end
```

DESIGN TWO:

```
class DATABASE_V2
feature {NONE} -- implementation
  imp: LIST[PERSON]
... -- more features and contracts
end
```

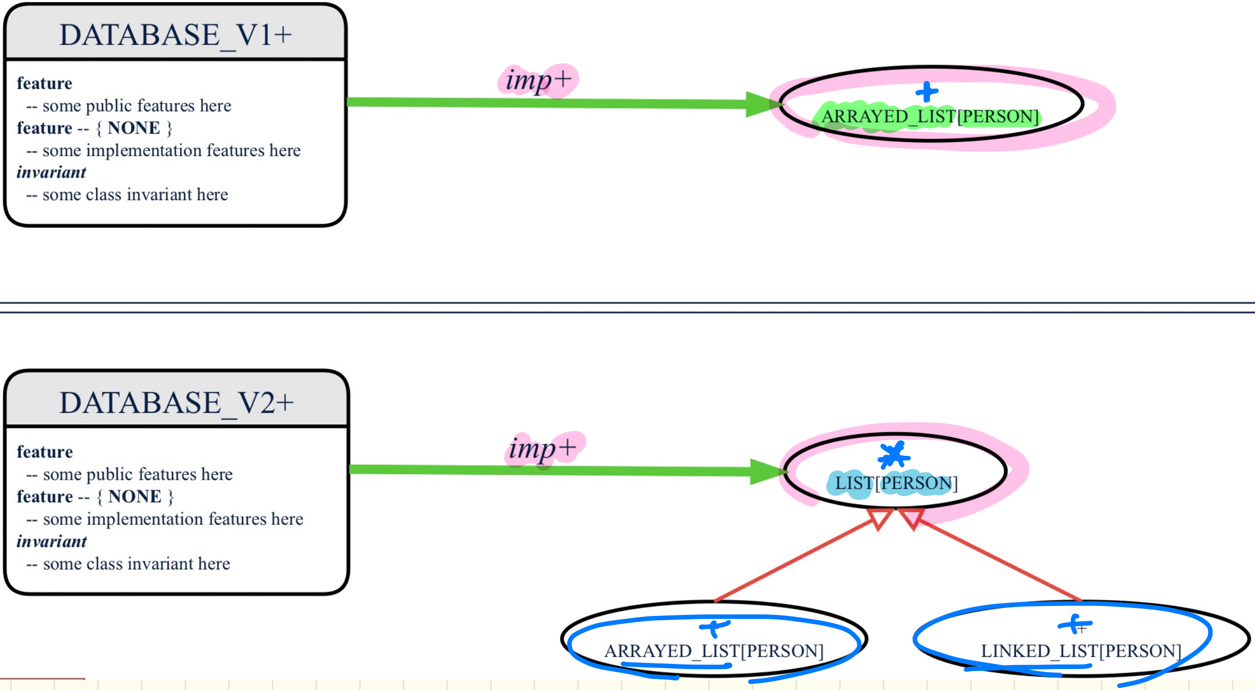
Presenting CS Relation in Diagram: Approach 1

We may focus on the PERSON supplier class, which may not help judge which design is better.



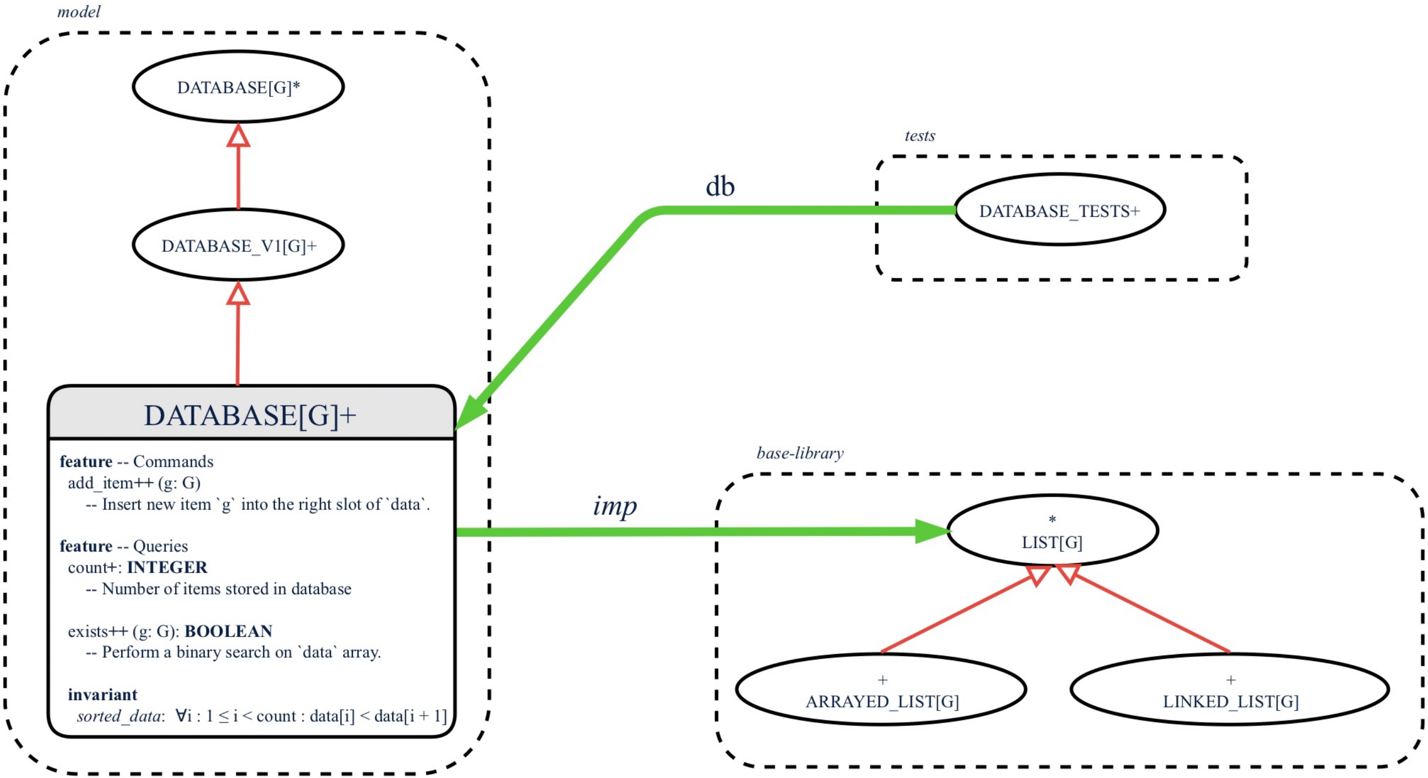
Presenting CS Relation in Diagram: Approach 2

Alternatively, we may focus on the `LIST` supplier class, which in this case helps us judge which design is better.



Clusters

Use *clusters* to group classes into logical units.



Lecture 4

Part 2

Postcondition: Asserting Set Equality

Writing Postcondition: Exercise

Problem

```
all_positive_values (a: ARRAY[INTEGER]): ARRAY[INTEGER]
    require
      no_duplicates: ??
    ensure
      across Result is x
        all
          x > 0
        end
```

all_p_vs (<< 2, 3, -1, 4, -2 >>)
↳ << 2, 3, 4 >>
Correct

incomplete

Witness/Counter-example

all_p_vs (<< 2, 3, -1, 4, -2 >>)
↳ << 1, 5, 7 >>

wrong imp.

↳ since each num. is > 0 ⇒ no violation postcond.

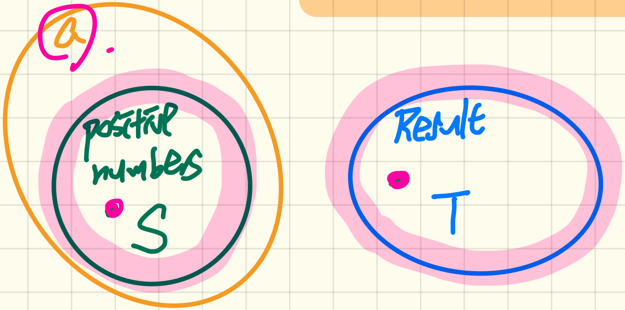
but the output is incorrect

design is flawed

Writing Postcondition: Exercise

Solution

```
all_positive_values (a: ARRAY[INTEGER]): ARRAY[INTEGER]
    require
      no_duplicates: ??
    ensure
      across Result is x
        all
          x > 0
        end
```



$$S = T \equiv (\forall x \mid x \in S \Rightarrow x \in T \;\; \wedge \;\; x \in T \Rightarrow x \in S)$$

$S \subseteq T$

$T \subseteq S$

```
all_pos_in_a_also_in_result:
    across a is x
    all x > 0 implies Result.has(x) end
```

```
all_num_in_result_pos_and_in_a:
    across Result is x
    all x > 0 and a.has(x) end
```

Lecture 4

Part 3

Abstraction of a Birthday Book

Testing REL in MATHMODELS

r.count → 9

Say $r = \{(a,1), (b,2), (c,3), (a,4), (b,5), (c,6), (d,1), (e,2), (f,3)\}$

r.domain → {a, b, c, d, e, f}

r.range → {1, 2, 3, 4, 5, 6}

r.image(a) → {1, 4} → function domain → at most 1 value in range

r.image(a).count = 1 → singleton set

a: domain value

r[g] → ∅

Say $r = \{(a,1), (b,2), (c,3), (a,4), (b,5), (c,6), (d,1), (e,2), (f,3)\}$

$r.\textbf{overridden}(\underbrace{\{(a,3),(c,4)\}}_{t})$

$= \underbrace{\{(a,3),(c,4)\}}_{t} \cup \underbrace{\{(b,2),(b,5),(d,1),(e,2),(f,3)\}}_{r.\textbf{domain\_subtracted}(\underbrace{t.\textbf{domain}}_{\{a,c\}})}$

$= \{(a,3),(c,4),(b,2),(b,5),(d,1),(e,2),(f,3)\}$

(a, 3)
(c, 4)

Testing REL in MATHMODELS

Command : imp. of model

Query : contracts

```
test_rel: BOOLEAN
  local
    r, t: REL[STRING, INTEGER]
    ds: SET[STRING]
  do
    create r.make_from_tuple_array (
      <<["a", 1], ["b", 2], ["c", 3],
        ["a", 4], ["b", 5], ["c", 6],
        ["d", 1], ["e", 2], ["f", 3]>>)
    create ds.make_from_array (<<"a">>)
    -- r is not changed by the query 'domain_subtracted'
    t := r.domain_subtracted (ds)
    Result :=
      t /~ r and not t.domain.has ("a") and r.domain.has ("a")
    check Result end
    -- r is changed by the command 'domain_subtract'
    r.domain_subtract (ds)
    Result :=
      t ~ r and not t.domain.has ("a") and not r.domain.has ("a")
  end
```

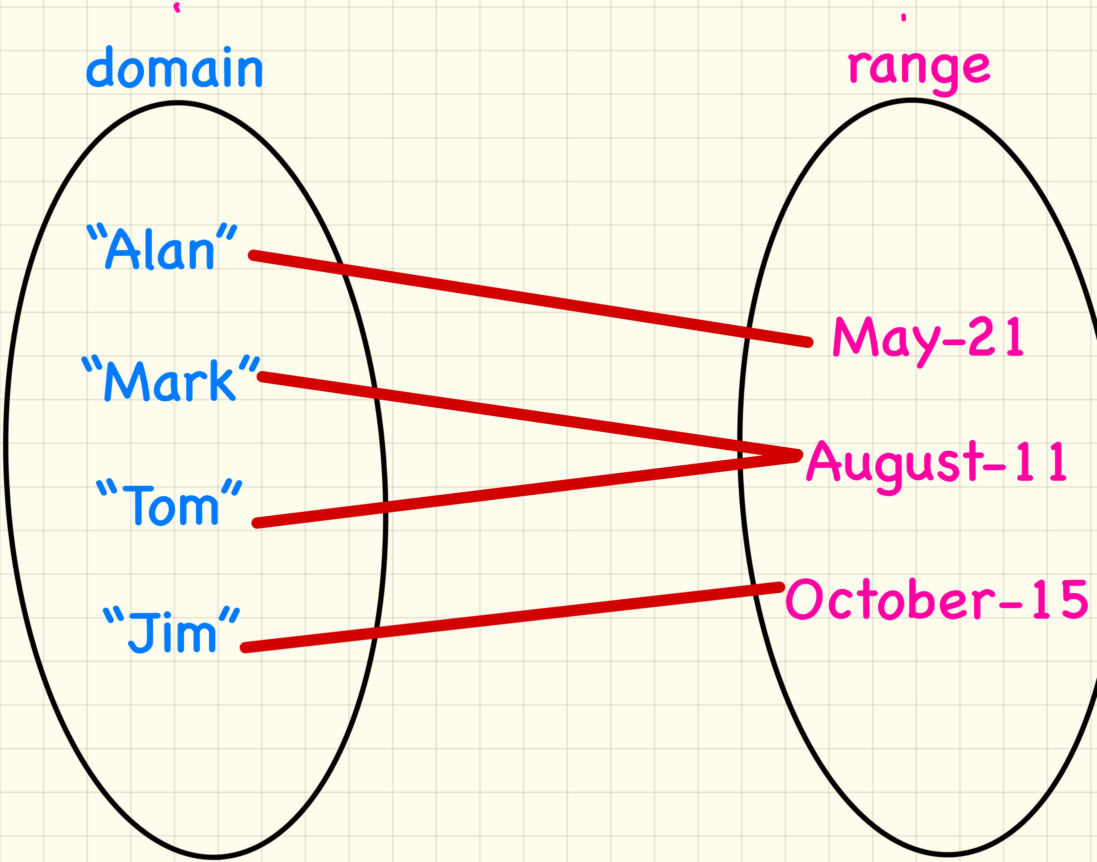
t: (b,2) (c,3) (b,5) (c,6) (d,1) (e,2) (f,3)

ds → {"a"}

Say $r = \{(a,1),(b,2),(c,3),(a,4),(b,5),(c,6),(d,1),(e,2),(f,3)\}$

- r.*domain*: set of first-elements from r
  - r.**domain** = $\{ d \mid (d,r) \in r \}$
  - e.g., r.**domain** = $\{a,b,c,d,e,f\}$
- r.*range*: set of second-elements from r
  - r.**range** = $\{ r \mid (d,r) \in r \}$
  - e.g., r.**range** = $\{1,2,3,4,5,6\}$
- r.*inverse*: a relation like r except elements are in reverse order
  - r.**inverse** = $\{ (r,d) \mid (d,r) \in r \}$
  - e.g., r.**inverse** = $\{(1,a),(2,b),(3,c),(4,a),(5,b),(6,c),(1,d),(2,e),(3,f)\}$
- r.*domain_restricted*(ds): sub-relation of r with domain ds.
  - r.**domain_restricted**(ds) = $\{ (d,r) \mid (d,r) \in r \wedge d \in ds \}$
  - e.g., r.**domain_restricted**({a, b}) = $\{(\mathbf{a},1),(\mathbf{b},2),(\mathbf{a},4),(\mathbf{b},5)\}$
- r.*domain_subtracted*(ds): sub-relation of r with domain not ds.
  - r.**domain_subtracted**(ds) = $\{ (d,r) \mid (d,r) \in r \wedge d \notin ds \}$
  - e.g., r.**domain_subtracted**({a, b}) = $\{(\mathbf{c},6),(\mathbf{d},1),(\mathbf{e},2),(\mathbf{f},3)\}$
- r.*range_restricted*(rs): sub-relation of r with range rs.
  - r.**range_restricted**(rs) = $\{ (d,r) \mid (d,r) \in r \wedge r \in rs \}$
  - e.g., r.**range_restricted**({1, 2}) = $\{(a,\mathbf{1}),(b,\mathbf{2}),(d,\mathbf{1}),(e,\mathbf{2})\}$
- r.*range_subtracted*(ds): sub-relation of r with range not ds.
  - r.**range_subtracted**(rs) = $\{ (d,r) \mid (d,r) \in r \wedge r \notin rs \}$
  - e.g., r.**range_subtracted**({1, 2}) = $\{(c,\mathbf{3}),(a,\mathbf{4}),(b,\mathbf{5}),(c,\mathbf{6})\}$

Model of an Example Birthday Book



Birthday Book: Design

BIRTHDAY_BOOK

```
model: FUN[NAME, BIRTHDAY]
  -- abstraction function

count: INTEGER
  -- number of entries

put(n: NAME, d: BIRTHDAY)
  ensure
    model_operation: model ~ (old model.deep_twin) overriden_by ([n,d])
    -- infix symbol for override operator: @<+

remind(d: BIRTHDAY): ARRAY[NAME]
  ensure
    nothing_changed: model ~ (old model.deep_twin)
    same_counts: Result.count = (model.range_restricted_by(d)).count
    same_contents: ∀ name ∈ (model.range_restricted_by(d)).domain: name ∈ Result
    -- infix symbol for range restriction: model @> (d)

invariant:
  consistent_book_and_model_counts: count = model.count
```

model: FUN[NAME, ..] → BIRTHDAY

BIRTHDAY

```
day: INTEGER
month: INTEGER

invariant
  1 ≤ month ≤ 12
  1 ≤ day ≤ 31
```

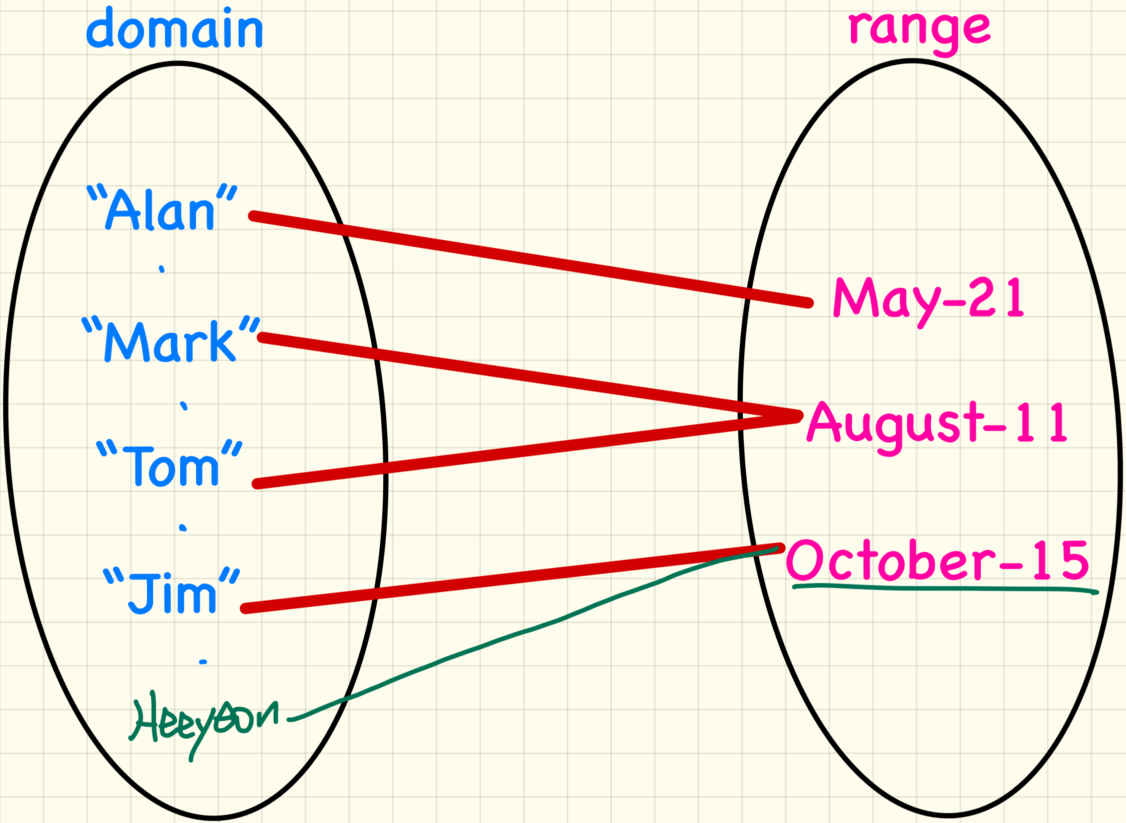
remind: ARRAY[..] → NAME

NAME

```
item: STRING

invariant
  item[1] ∈ A..Z
```

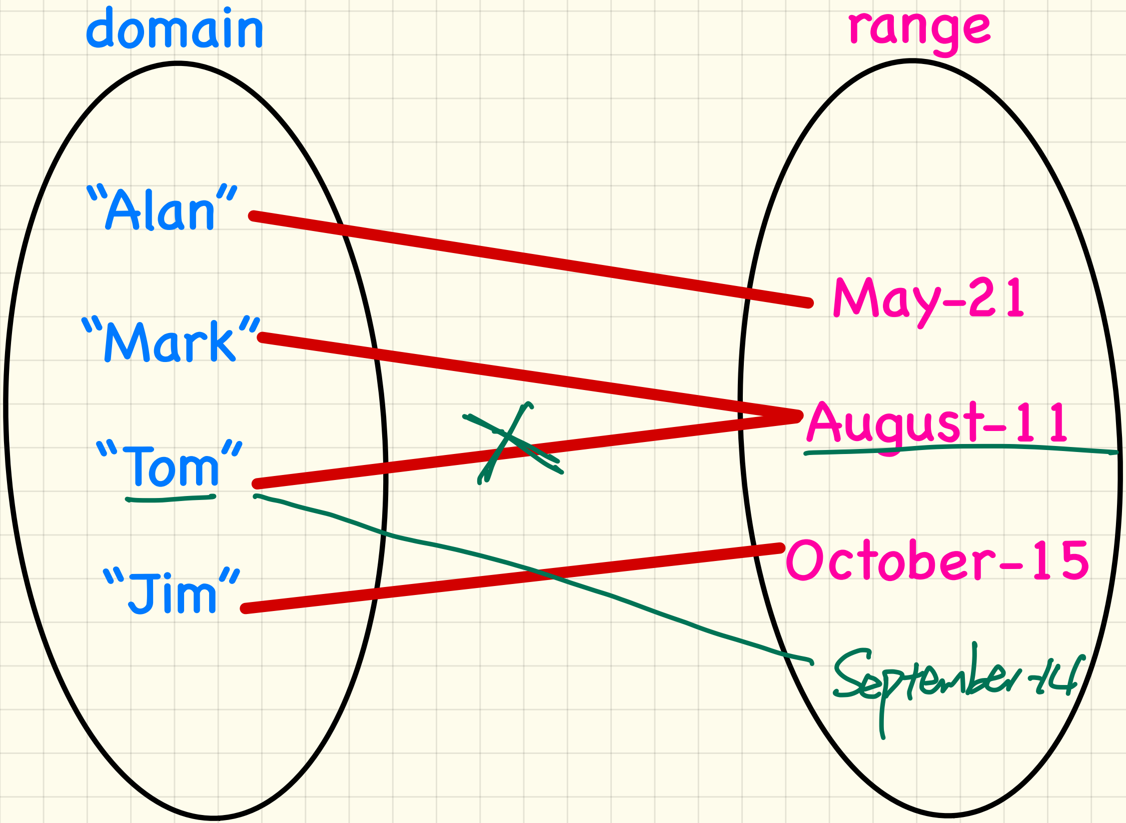
Birthday Book: Model Operation (1.1)

book.put("Heeyeon", October-15)



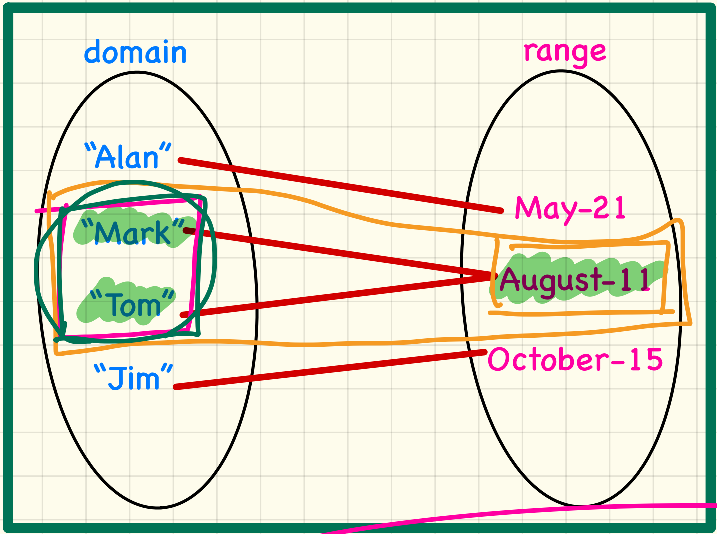
Birthday Book: Model Operation (1.2)

book.put("Tom", September-14)



Birthday Book: Model Operation (2.1)

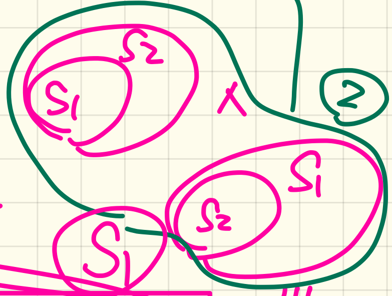
book.remind(August-11)



$\#S_1 = \#S_2 \land S_1 \subseteq S_2$

$S_1 = S_2$

↳ $S_1 \subseteq S_2$, $S_2 \subseteq S_1$



set

(model.range_restricted_by(August-11)).domain

relation

~ Result

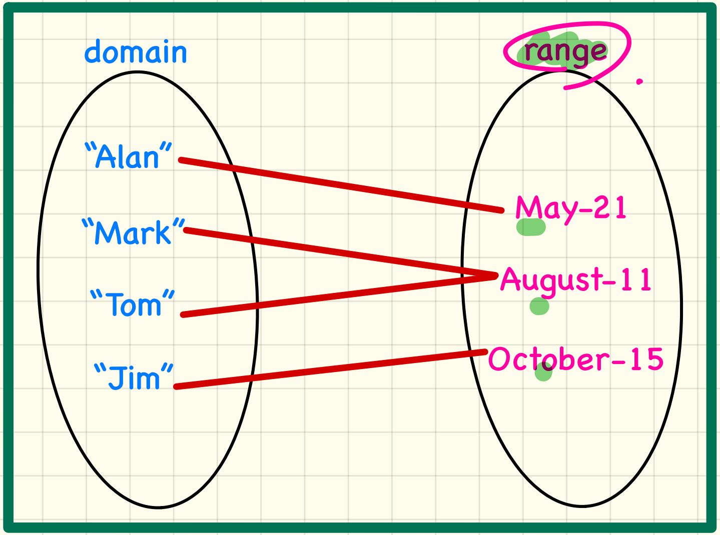
Array

S_2

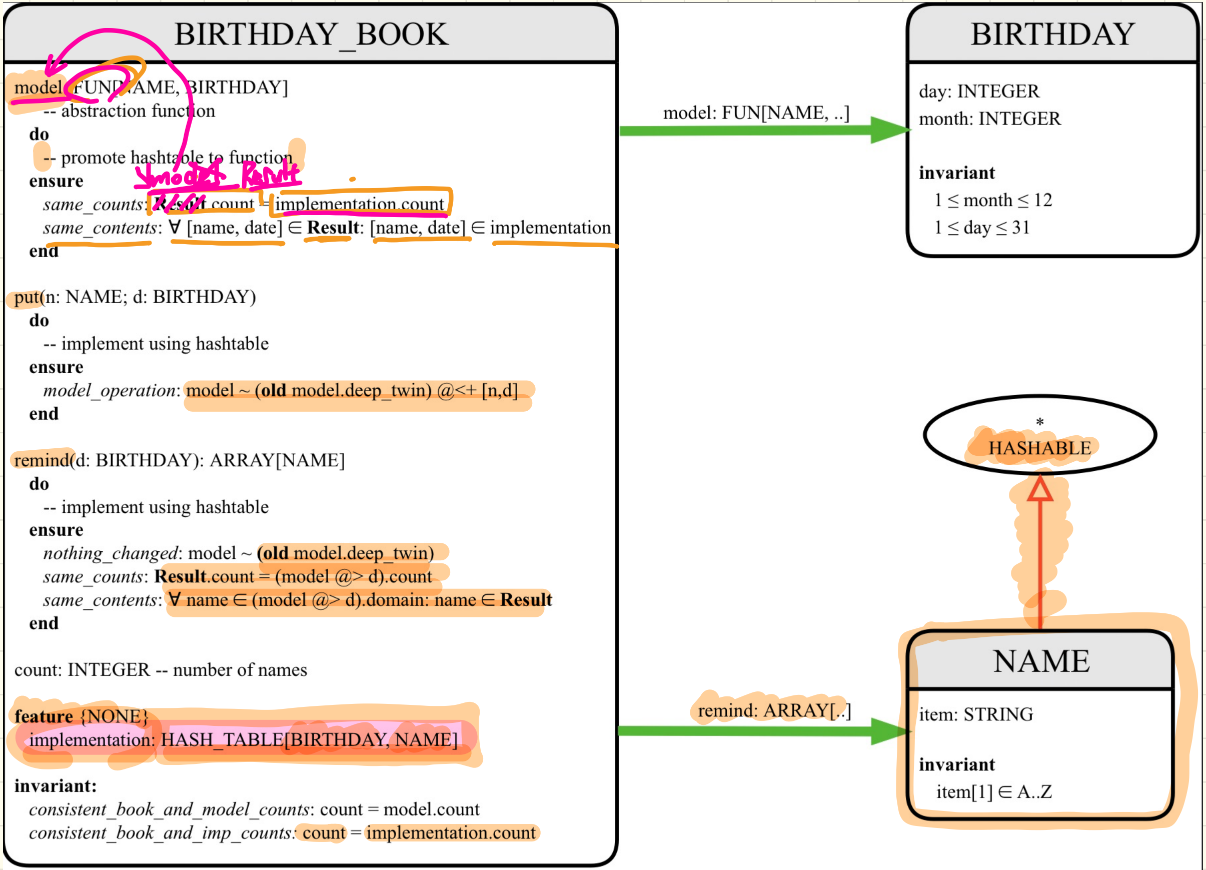
Birthday Book: Model Operation (2.2)

book.remind(November-29)

↳ ~~empty array~~.



Birthday Book: Implementation



Lecture 4

Part 4

Design Pattern: Iterator

Principle of **Information Hiding**

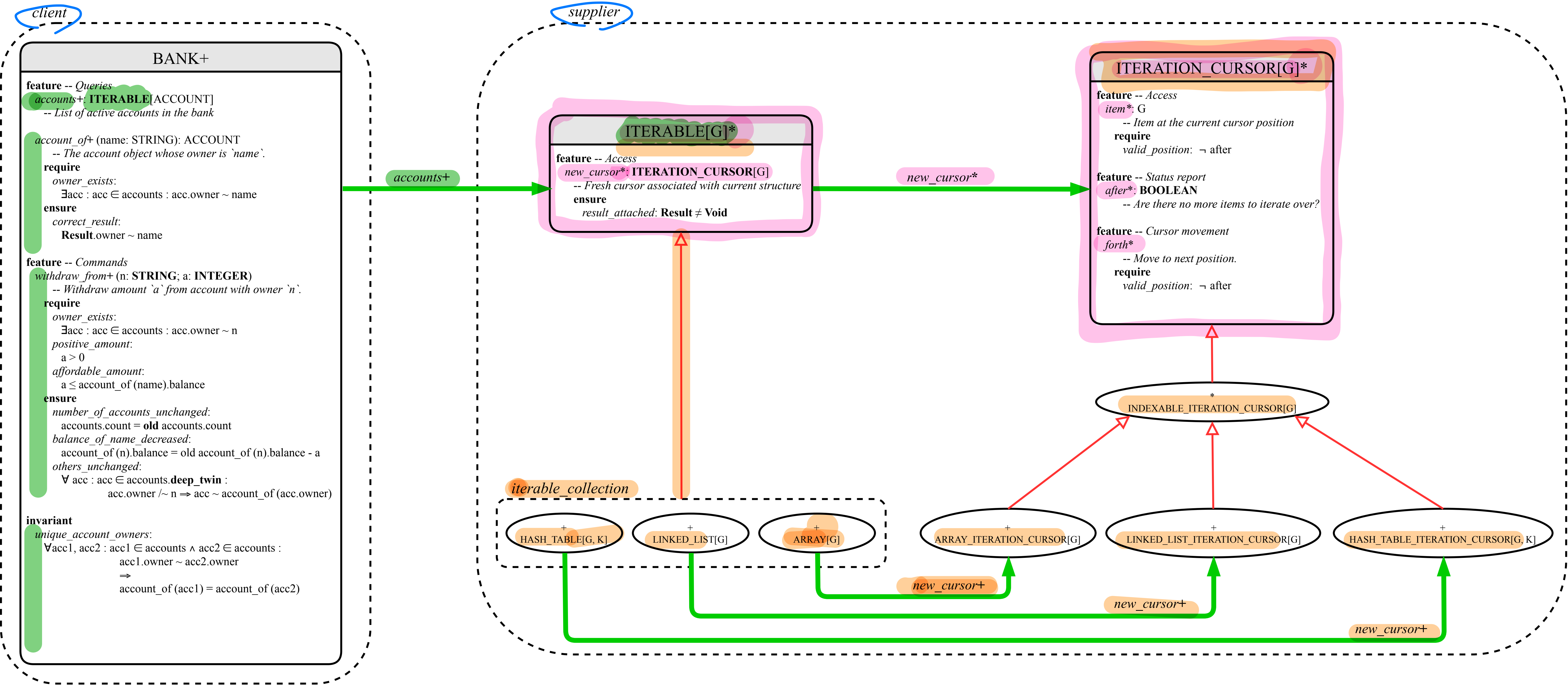
Supplier:

```
class
  CART
feature
  orders: ARRAY[ORDER]
end

class
  ORDER
feature
  price: INTEGER
  quantity: INTEGER
end
```

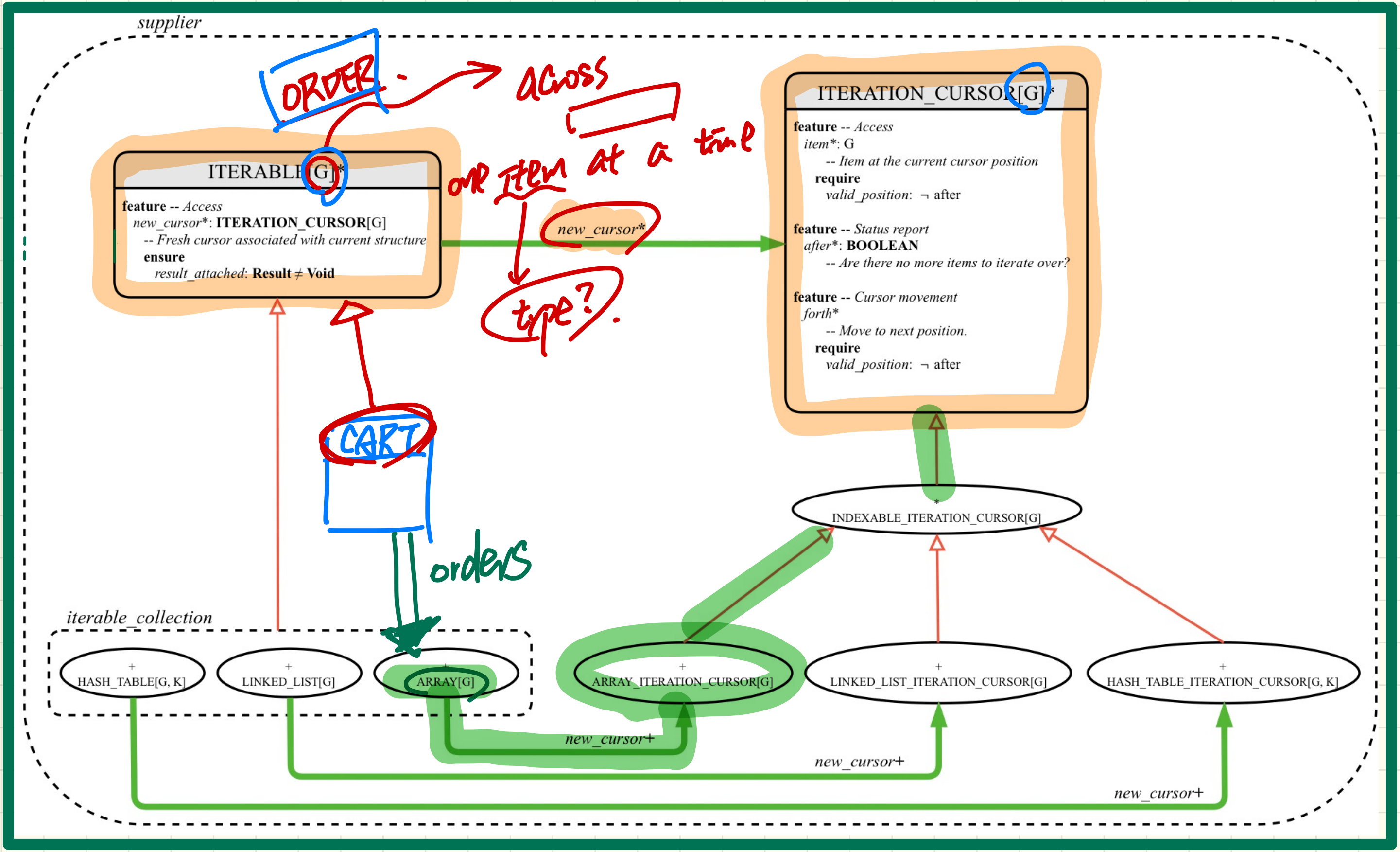
Client:

```
class
  SHOP
feature
  cart: CART
  checkout: INTEGER
    do
      from
        i := cart.orders.lower
      until
        i > cart.orders.upper
      do
        Result := Result +
          cart.orders[i].price
          *
          cart.orders[i].quantity
        i := i + 1
      end
    end
end
```

Problems?


client
supplier
BANK+
feature -- Queries
accounts+: ITERABLE[ACCOUNT]
-- List of active accounts in the bank
account_of+ (name: STRING): ACCOUNT
-- The account object whose owner is `name`.
require
owner_exists:
∃acc : acc ∈ accounts : acc.owner ~ name
ensure
correct_result:
Result.owner ~ name
feature -- Commands
withdraw_from+ (n: STRING; a: INTEGER)
-- Withdraw amount `a` from account with owner `n`.
require
owner_exists:
∃acc : acc ∈ accounts : acc.owner ~ n
positive_amount:
a > 0
affordable_amount:
a ≤ account_of (name).balance
ensure
number_of_accounts_unchanged:
accounts.count = old accounts.count
balance_of_name_decreased:
account_of (n).balance = old account_of (n).balance - a
others_unchanged:
∀ acc : acc ∈ accounts.deep_twin :
acc.owner /~ n ⇒ acc ~ account_of (acc.owner)
invariant
unique_account_owners:
∀acc1, acc2 : acc1 ∈ accounts ∧ acc2 ∈ accounts :
acc1.owner ~ acc2.owner
⇒
account_of (acc1) = account_of (acc2)
accounts+
ITERABLE[G]*
feature -- Access
new_cursor*: ITERATION_CURSOR[G]
-- Fresh cursor associated with current structure
ensure
result_attached: Result ≠ Void
new_cursor*
ITERATION_CURSOR[G]*
feature -- Access
item*: G
-- Item at the current cursor position
require
valid_position: ¬ after
feature -- Status report
after*: BOOLEAN
-- Are there no more items to iterate over?
feature -- Cursor movement
forth*
-- Move to next position.
require
valid_position: ¬ after
*
INDEXABLE_ITERATION_CURSOR[G]
iterable_collection
+
HASH_TABLE[G, K]
+
LINKED_LIST[G]
+
ARRAY[G]
+
ARRAY_ITERATION_CURSOR[G]
+
LINKED_LIST_ITERATION_CURSOR[G]
+
HASH_TABLE_ITERATION_CURSOR[G, K]
new_cursor+
new_cursor+
new_cursor+

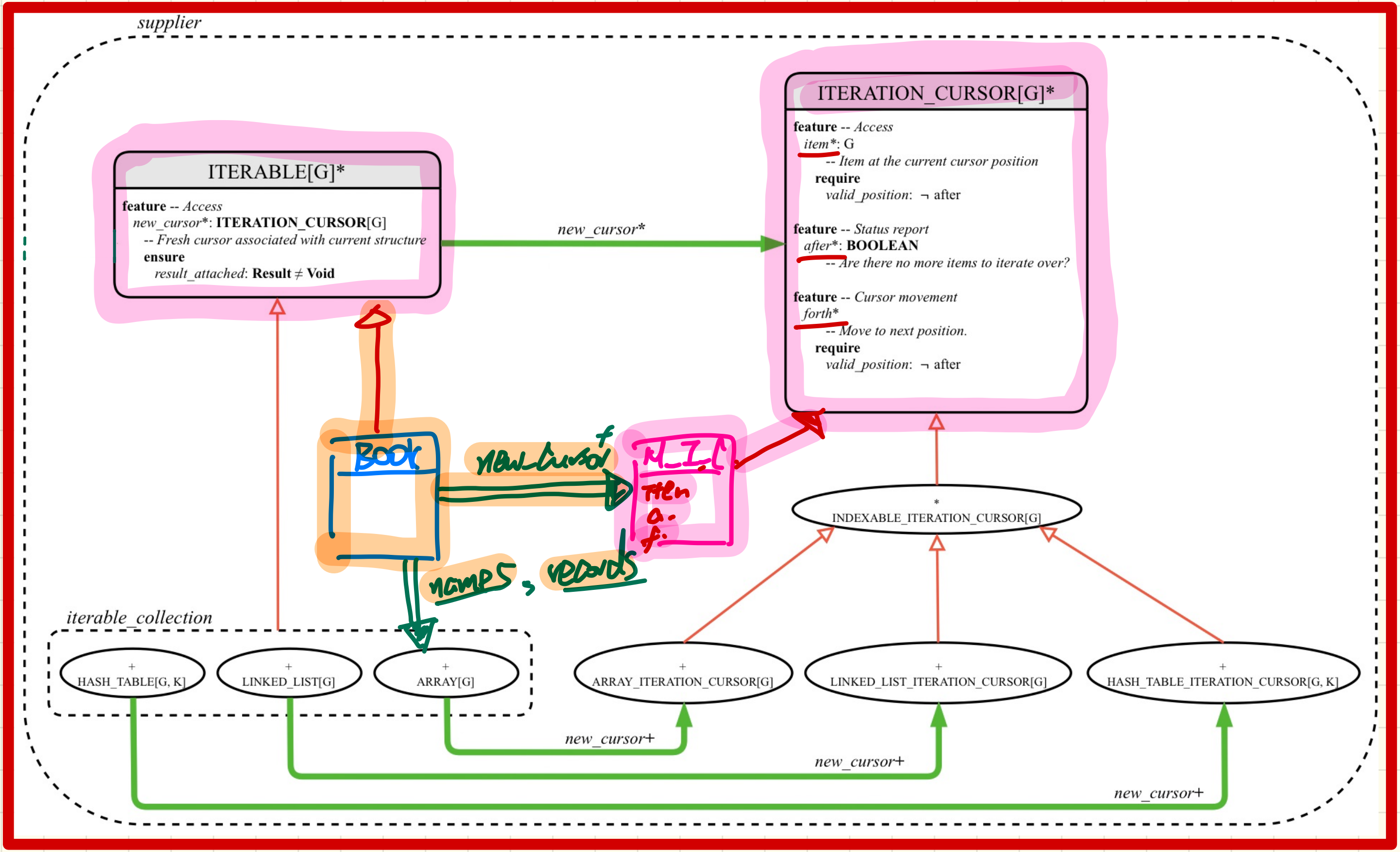
Implementing the Iterator Pattern: Easy Case



Implementing the Iterator Pattern: Easy Case

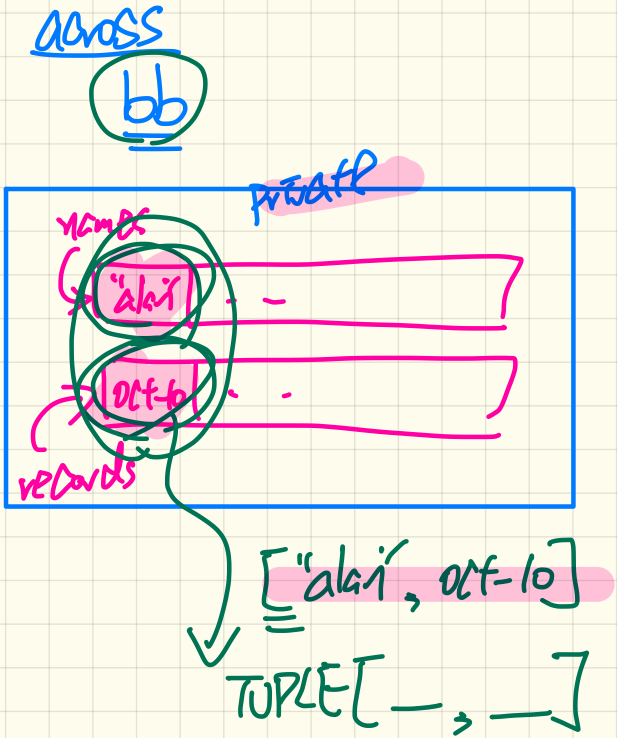
```
class
  CART
inherit
    ITERABLE[ORDER]
feature {NONE}
  orders: ARRAY[ORDER]

feature --
    new_cursor : I_C[ORDER]
      do
        Result := orders.new_cursor
      end
end
```

Implementing the Iterator Pattern: Hard Case

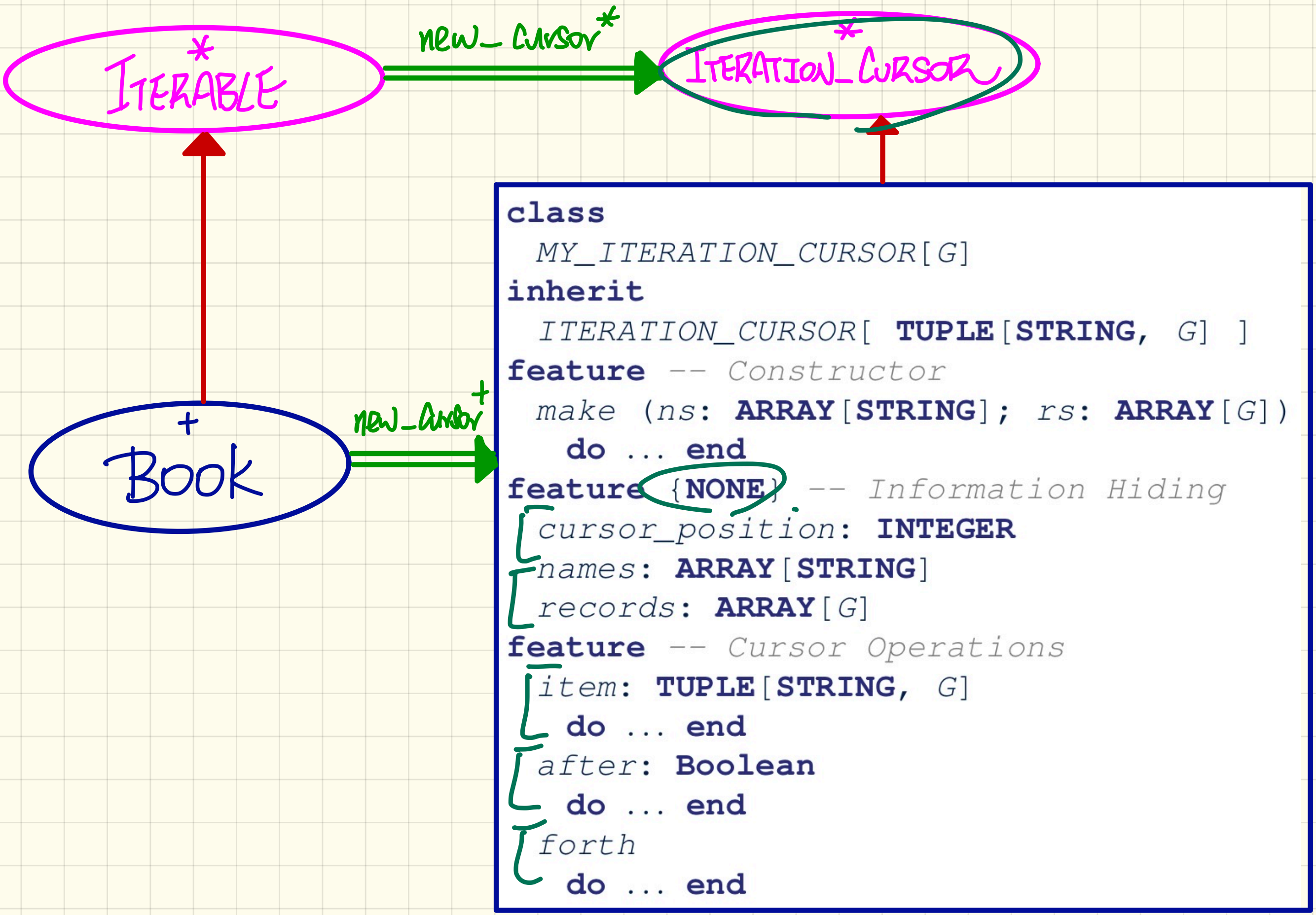


Implementing the Iterator Pattern: Hard Case

```
class
  BOOK[G]
feature {NONE}
  names: ARRAY[STRING]
  records: ARRAY[G]
end
```

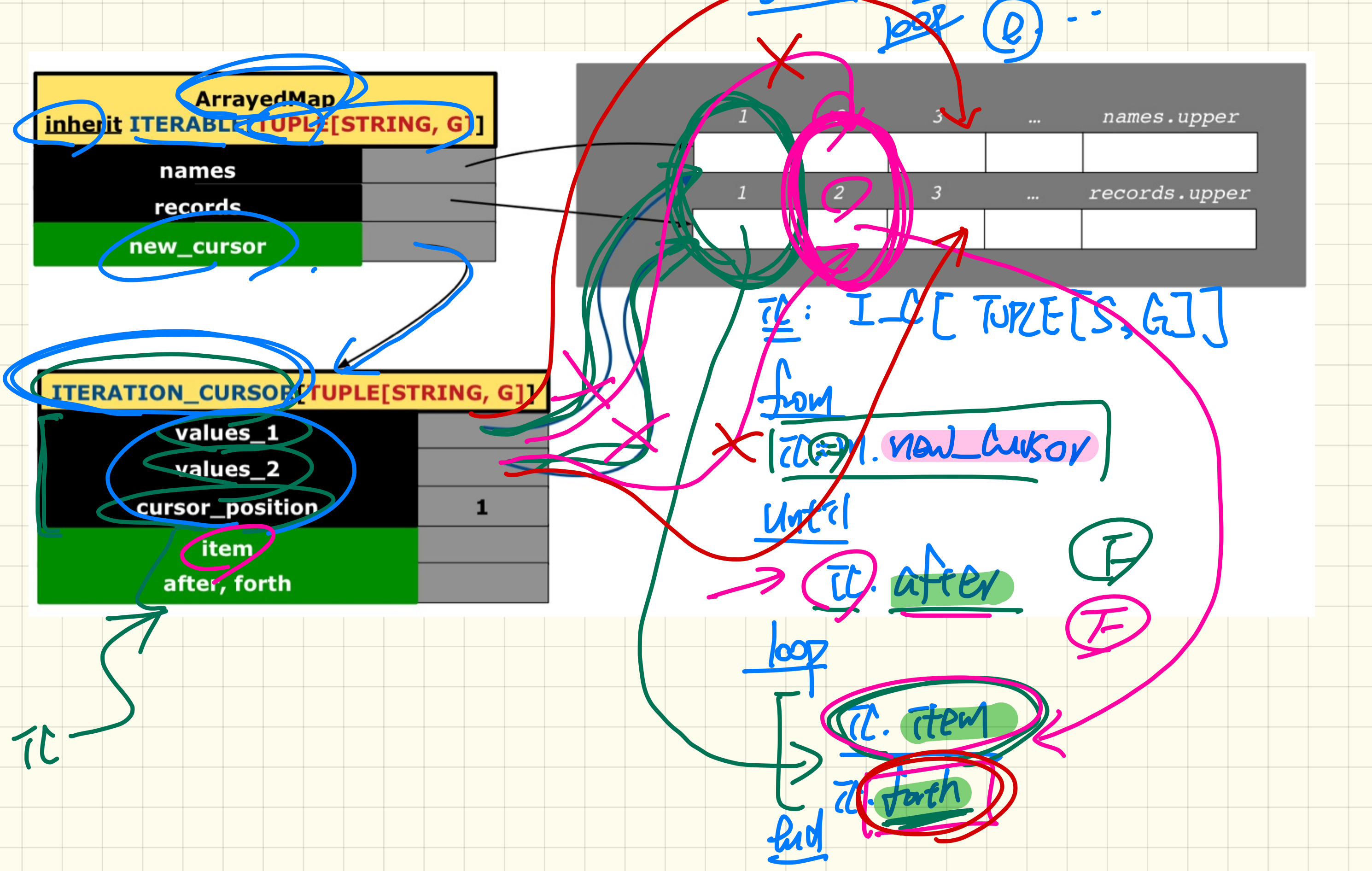


Implementing the Iterator Pattern: Hard Case




```
class
  MY_ITERATION_CURSOR[G]
inherit
  ITERATION_CURSOR[ TUPLE[STRING, G] ]
feature -- Constructor
  make (ns: ARRAY[STRING]; rs: ARRAY[G])
    do ... end
feature {NONE} -- Information Hiding
  cursor_position: INTEGER
  names: ARRAY[STRING]
  records: ARRAY[G]
feature -- Cursor Operations
  item: TUPLE[STRING, G]
    do ... end
  after: Boolean
    do ... end
  forth
    do ... end
```

Iterator Pattern at Runtime



Use of Iterable in Contracts

```
class
  CHECKER
feature -- Attributes
  collection: ITERABLE [INTEGER]
feature -- Queries
  is_all_positive: BOOLEAN
      -- Are all items in collection positive?
    do
      ...
    ensure
      across
        collection is item
      all
        item > 0
      end
  end
```

```
class BANK
...
  accounts: LIST [ACCOUNT]
  binary_search (acc_id: INTEGER): ACCOUNT
      -- Search on accounts sorted in non-descending order.
    require
      across
        1 |..| (accounts.count - 1) is i
      all
        accounts [i].id <= accounts [i + 1].id
      end
    do
      ...
    ensure
      Result.id = acc_id
    end
```

ITERABLE ✗ does not support [..]

item ↳ specific to LIST

item

Use of Iterable in Contracts: Exercise

```
class BANK
...
  accounts: LIST [ACCOUNT]
  contains_duplicate: BOOLEAN
      -- Does the account list contain duplicate?
    do
      ...
    ensure
```

$$\forall i, j : INTEGER \mid 1 \leq i \leq accounts.count \wedge 1 \leq j \leq accounts.count \bullet accounts[i] \sim accounts[j] \Rightarrow i = j$$

```
    end
```

across

Use of Iterable in Implementation (1)

```
class BANK
  accounts: ITERABLE [ACCOUNT]
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      cursor: ITERATION_CURSOR[ACCOUNT]; max: ACCOUNT
    do
      from cursor := accounts.new_cursor; max := cursor.item
      until cursor.after
      do
        if cursor.item.balance > max.balance then
          max := cursor.item
        end
        cursor.forth
      end
    ensure ??
    end
```

loop

Use from-until loop if you wish to work with an I.C.

Use of Iterable in Implementation (2)

```
class SHOP
  cart: CART
  checkout: INTEGER
      -- Total price calculated based on orders in the cart.
    require ??
    do
      across
        cart is order
      loop
        Result := Result + order.price * order.quantity
      end
    ensure ??
  end
```

TC := cart.new_cursor

TC.item.

TC.item.

TC.forth

```
class BANK
  accounts: LIST[ACCOUNT] -- Q: Can ITERABLE[ACCOUNT] work?
  max_balance: ACCOUNT
      -- Account with the maximum balance value.
    require ??
    local
      max: ACCOUNT
    do
      max := accounts [1]
      across
        accounts is acc
      loop
        if acc.balance > max.balance then
          max := acc
        end
      end
    ensure ??
  end
```

ITERABLE ?

Lecture 4

Part 5

Exercise: Generics in Iterator

Exercise

```
deferred class
  ITERABLE [G]
feature -- Access
  new_cursor: ITERATION_CURSOR [G]
    deferred end
end
```

new_cursor*

```
deferred class
  ITERATION_CURSOR [G]
feature -- Cursor features
  item: G
    deferred end

  after: BOOLEAN
    deferred end

  forth
    deferred end
end
```

```
test_database: BOOLEAN
  local
    db: DATABASE[STRING, INTEGER]
    tuples: LINKED_LIST[TUPLE[INTEGER, STRING]]
  do
    create db.make
    create tuples.make
    across
      db is t
    loop
      tuples.extend (t)
    end
end
```

db+

```
class
  DATABASE[G, H]
inherit
  ITERABLE[ ___ ]
feature {NONE} -- Implementation
  gs: ARRAY[G]
  hs: ARRAY[H]
feature -- Iterable
  new_cursor: ITERATION_CURSOR[ ___ ]
    local
      db_cursor: ITEM_ITERATION_CURSOR[H, G]
    do
      create db_cursor.make ( ___ )
      Result := db_cursor
    end
end
```

new_cursor+

```
class
  ITEM_ITERATION_CURSOR[M, N]
inherit
  ITERATION_CURSOR[ ___ ]
create
  make
feature {NONE} -- Implementation
  ms: ARRAY[M]
  ns: ARRAY[N]
feature -- Constructor
  make (new_ns: ARRAY[N]; new_ms: ARRAY[M])
    do ... end
feature -- Cursor features
  item: ___
    do ... end

  after: BOOLEAN
    do ... end

  forth
    do ... end
end
```