EEC S3311 Software Design (Fall 2020)

Q&A - Lecture Series W5

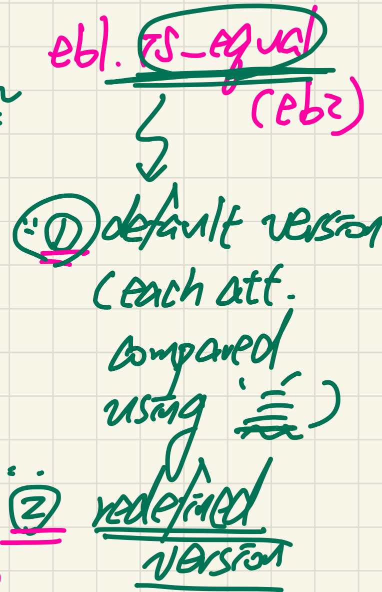
Tuesday, October 20

When comparing two expanded classes, "=" and "~" are exactly the same?

Let's say x and y are instances of the same class, and within that class there is some reference type attribute called `r`. When we do `x = y`, are we doing a reference comparison of all the attributes (e.g., `x.r = y.r`)?

And when we do `x ~ y`, we're doing a content comparison (e.g., `x.r ~ y.r`)?

Or is it the case that when we compare two expanded objects of the same type, we're always comparing contents, regardless of the equal symbol? So even when we do `x = y`, it'll just be `x.r ~ y.r`?



Expanded Class vs Deep Copying (4)

Equality and Dynamic Binding

```
expanded class
  B1
inherit
  ANY
    redefine
      default_create
    end
create
  default_create
feature -- command
  default_create
    do
      create s.make_empty
    end
feature
  i: INTEGER
  s: STRING
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
  change_s (ns: STRING)
    do
      s := ns
    end
end
```

```
expanded class
  B2
inherit
  ANY
    redefine
      is_equal,
      default_create
    end
create
  default_create
feature -- command
  default_create
    do
      create s.make_empty
    end
feature
  i: INTEGER
  s: STRING
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
  change_s (ns: STRING)
    do
      s := ns
    end
feature
  is_equal (other: like Current): BOOLEAN
    do
      Result := i = other.i and s ~ other.s
    end
end
```

```
local
  eb1, eb2: B1
  eb3, eb4: B2
  b1,b2: BOOLEAN
do
  ...
  eb1 = eb2
  eb3 ~ eb4
  ...
end
```

≡ eb1.is_equal(eb2) — default version "="

≡ eb3 = eb4

≡ eb3.is_equal(eb4)

Expanded Class vs Deep Copying (5)

Incompatible Types

```
expanded class
  B1
inherit
  ANY
    redefine
      default_create
    end
create
  default_create
feature -- command
  default_create
    do
      create s.make_empty
    end
feature
  i: INTEGER
  s: STRING
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
  change_s (ns: STRING)
    do
      s := ns
    end
end
```

```
expanded class
  B2
inherit
  ANY
    redefine
      is_equal,
      default_create
    end
create
  default_create
feature -- command
  default_create
    do
      create s.make_empty
    end
feature
  i: INTEGER
  s: STRING
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
  change_s (ns: STRING)
    do
      s := ns
    end
feature
  is_equal (other: like Current): BOOLEAN
    do
      Result := i = other.i and s ~ other.s
    end
end
```

```
test_expanded_equal: BOOLEAN
  local
    eb1, eb2: B1   -> default
    eb3, eb4: B2   -> redefined.
  do
    -- put a breakpoint here and step into: you'll {ANY}.is_equal
    Result := eb1 /= eb2
    check Result end

    -- put a breakpoint here and step into: you'll {B2}.is_equal
    Result := eb3 = eb4
    check Result end

    -- put a breakpoint and step into: you get an immediate false because
    -- objects of different types are being compared.
    Result := eb1 /= eb3
    check Result end
  end
```

eb1 = eb3 → trivially false at runtime (not calling any version of is_equal).

eb1 ↓ B1; eb3 ↓ B2

= vs. is_equal

① expanded

= ≡ ~ ≡ is_equal

↳ ① default version ("=" to compare each att)

② redefined version

When you define a class, if objects of that class may be compared using ~, then always redefine is_equal.

② reference

= : compare by reference

~ ≡ is_equal → ① default version ("=" to compare att)

② redefined version

Java. the default version obj1.equals(obj2) "obj1 == obj2"

Use as a Constructor but Not a Command

```
class CLIENT_2
  ...
  test: BOOLEAN
    local
      s, old_s: SUPPLIER
    do
      create s.make (5)
      old_s := s
      create s.make (5)
      print (old_s = s)
      old_s := s
      s.make (7)
      print (old_s = s)
    end
end
```

```
class SUPPLIER

create {CLIENT_1}
  make

feature {CLIENT_2}
  make (init_i: INTEGER)
    do
      i := init_i
    end

feature
  i: INTEGER
end
```

Handwritten notes: class CLIENT_1 test: BOOLEAN ; ✓ create s.make(…) ✗ s.make(7) — in isolation this call is ok. create make do end ✗ syntax error; last id's only. Being able to use 'make' as a constructor does not require that you're able to use it as a command. You can think this way.

Under the command version of `make`, there is a "do end" statement. We can't have "do end" for constructor `make` because Eiffel doesn't allow that.

So does constructor `make` just look up the "do end" from command `make` and then using that, initialize a new SUPPLIER object? But how can CLIENT_1 initialize a new SUPPLIER object when only CLIENT_2 has access to the command `make`?

Command `make` contains the implementation, i.e., the "do end".

Or, is it the case that when we have the constructor `make`:

```
create  make      -- constructor (shorthand, visible)
```

The above is really shorthand for the line below (hidden from user):

```
make (init_i: INTEGER) do  i := init_i  end      -- constructor (full ver. but hidden)
```

We can avoid initializing the object as expanded classes do it by default.

However, I was wondering in a class where we can have multiple constructors (some classes have make_empty and make_from_tuple) , how will expanded classes work in that case or are they allowed to have multiple constructors?

Implicitly, an expanded object obj is initialized as:

```
create obj.d_c
```

```
expanded class A
  -- implicitly: create d_c.
  i: INTEGER
```

```
expanded class B
create
  default_create
default_create
  do i := 5 end
i: INTEGER
```

```
expanded class C
create default_create,
  make
default_create do i := 5 end
make (ni : INT) do i := ni end
```

```
create oc.make(8)
```

```
oa: A    | oa.i = 0
         |   (d_c not redef.)
ob: B    | ob.i = 5
oc: C    | oc.i = 5;
```

Use of Distinct DATA_ACCESS Objects

Supplier:

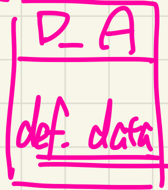
```
class BANK_DATA
create {BANK_DATA_ACCESS} make
feature {BANK_DATA_ACCESS}
  make do ... end
feature -- Data Attributes
  interest_rate: REAL
  set_interest_rate (r: REAL)
  ...
end
```

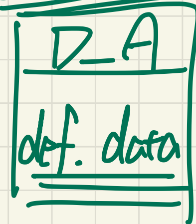
```
expanded class
  BANK_DATA_ACCESS
feature
  data: BANK_DATA
      -- The one and only access
    once create Result.make end
invariant data = data
```

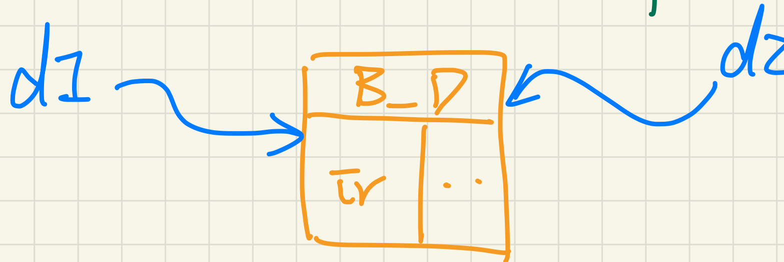
```
access, access_other: DATA_ACCESS
d1,d2 : DATA
...
d1 := access.data
d2 := access_other.data
```

1st call

2nd call







```
expanded class D_A
    data: B_D
      once ... end
end
```

```
expanded class D_A_CHILD
end
```

d1 → B_D
d2 → B_D

```
local
    a1: D_A
    a2: D_A_CHILD
    d1, d2: B_D
do
    d1 := a1.data
    d2 := a2.data
```

2nd call

a1: D_A — def. of data

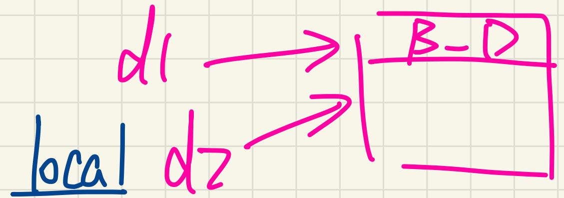
a2: D_A — def. of data

EXERCISE.

```
expanded class D_A
    data: B_D
    once ______ end
end
```

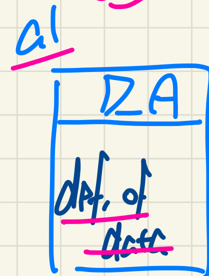
Q. still the same object?

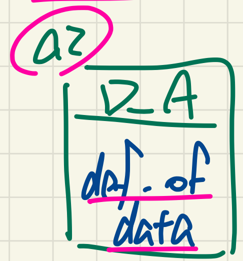
```
expanded class D_A_CHILD
    redefine data end
    data: B_D once ______ end.
end
```



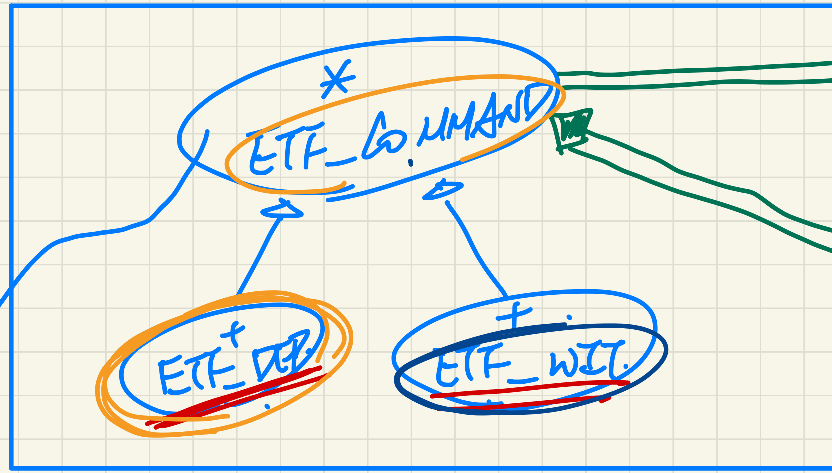

```
local
    a1: D_A
    a2: D_A_CHILD
    d1, d2: B_D
do
    d1 := a1.data
    d2 := a2.data
```

2nd call



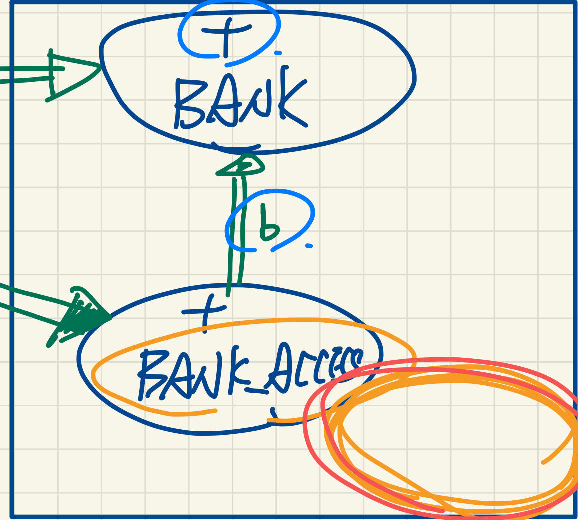


user_commands

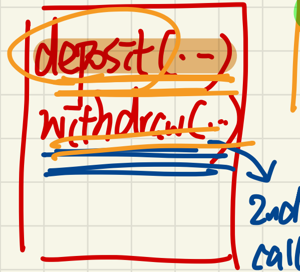


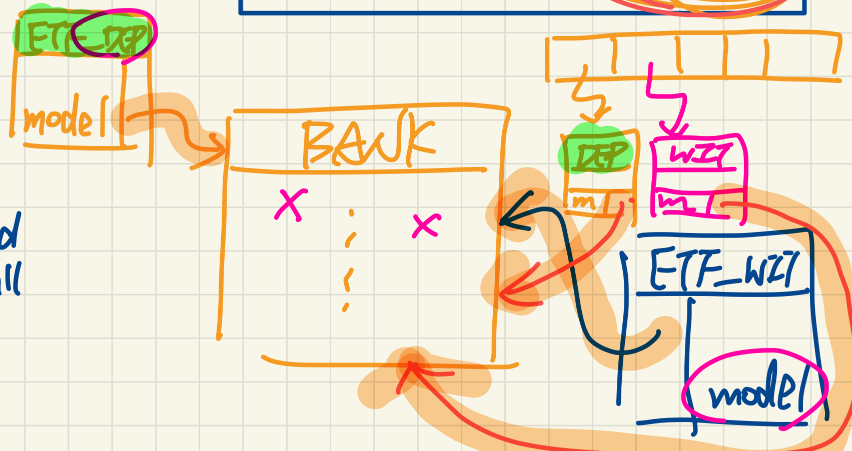
model

model




```
model : BANK
make
  local
    ba : BANK_ACCESS
  do
    model := ba.b
  end
```

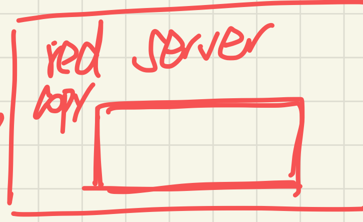




× SCP.

remotelab ×

github ×



sftp

Mac terminal

Win PuTTy.