Singleton Design Pattern

EECS3311 A & E: Software Design
Fall 2020

CHEN-WEI WANG

Learning Objectives

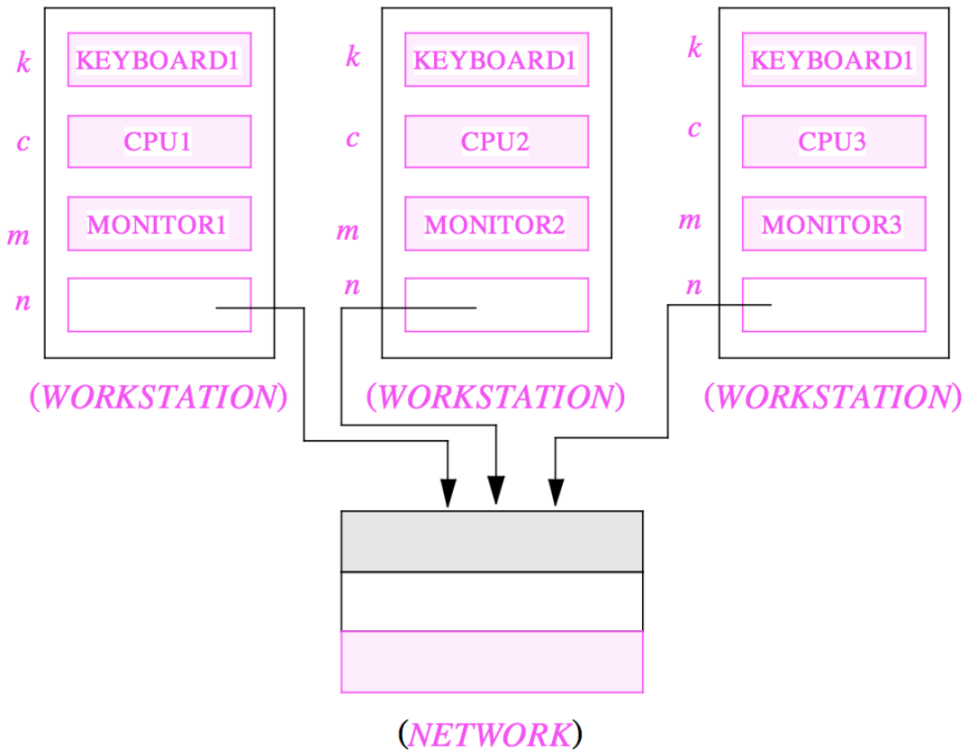
Upon completing this lecture, you are expected to understand:

1. Modeling Concept of ***Expanded Types*** (Compositions)
2. ***Once Routines*** in Eiffel vs. Static Methods in Java
3. Export Status
4. Sharing via ***Inheritance*** (w.r.t. ***SCP*** and ***Cohesion***)
5. *Singleton* Design Pattern

Expanded Class: Modelling

- We may want to have objects which are:
  - Integral parts of some other objects
  - ***Not*** shared among objects

  e.g., Each workstation has its own CPU, monitor, and keyword. All workstations share the same network.



Expanded Class: Programming (2)

```
class KEYBOARD ... end class CPU ... end
class MONITOR ... end class NETWORK ... end
class WORKSTATION
  k: expanded KEYBOARD
  c: expanded CPU
  m: expanded MONITOR
  n: NETWORK
end
```

Alternatively:

```
expanded class KEYBOARD ... end
expanded class CPU ... end
expanded class MONITOR ... end
class NETWORK ... end
class WORKSTATION
  k: KEYBOARD
  c: CPU
  m: MONITOR
  n: NETWORK
end
```

Expanded Class: Programming (3)

```
expanded class
  B
feature
  change_i (ni: INTEGER)
    do
      i := ni
    end
feature
  i: INTEGER
end
```

```
1   test_expanded
2    local
3      eb1, eb2: B
4    do
5      check eb1.i = 0 and eb2.i = 0 end
6      check eb1 = eb2 end
7      eb2.change_i (15)
8      check eb1.i = 0 and eb2.i = 15 end
9      check eb1 /= eb2 end
10     eb1 := eb2
11     check eb1.i = 15 and eb2.i = 15 end
12     eb1.change_i (10)
13     check eb1.i = 10 and eb2.i = 15 end
14     check eb1 /= eb2 end
15   end
```

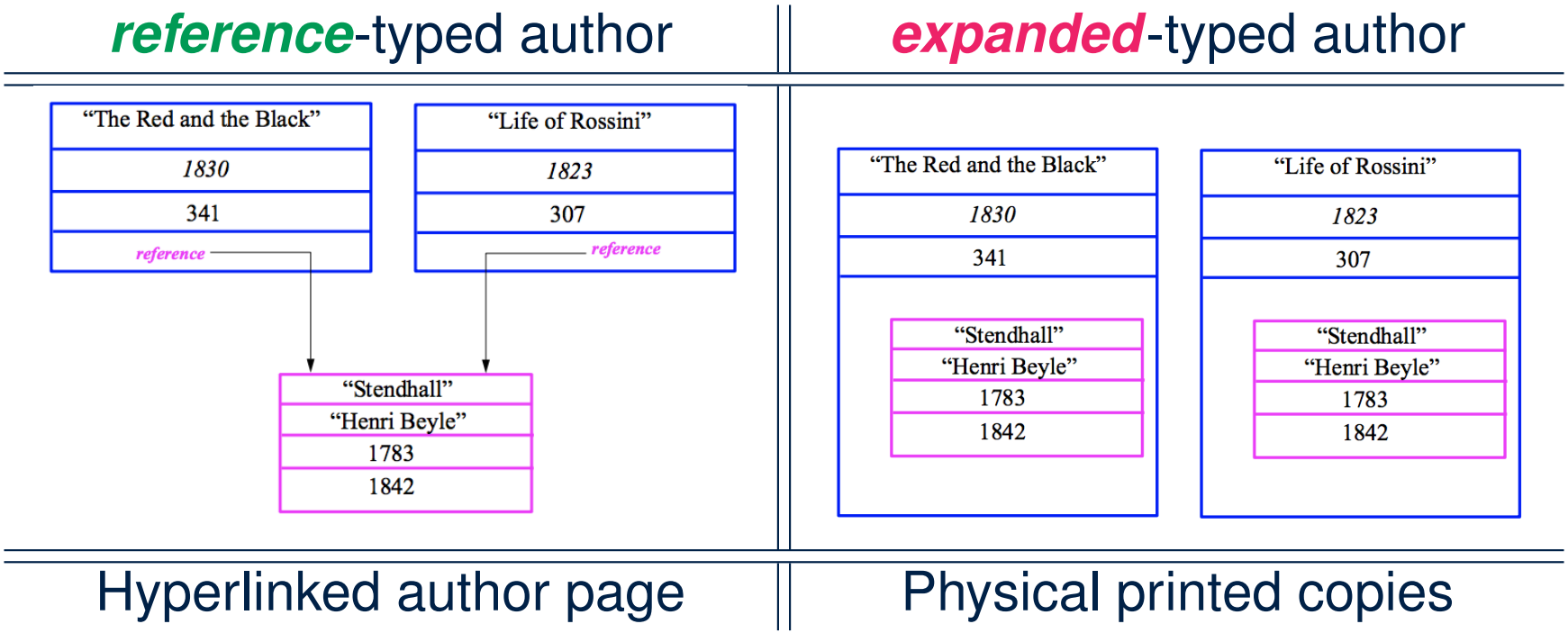
- **L5**: object of expanded type is automatically initialized.
- **L10,L12,L13**: no sharing among objects of expanded type.
- **L6,L9,L14**: = compares contents between expanded objects.

Reference vs. Expanded (1)

- Every entity must be declared to be of a certain type (based on a class).
- Every type is either ***referenced*** or ***expanded***.
- In ***reference*** types:
  - `y` denotes ***a reference*** to some object
  - `x := y` attaches `x` to same object as does `y`
  - `x = y` compares references
- In ***expanded*** types:
  - `y` denotes ***some object*** (of expanded type)
  - `x := y` copies contents of `y` into `x`
  - `x = y` compares contents `[x ~ y]`

Reference vs. Expanded (2)

Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book ***reference***-typed or ***expanded***-typed?



Singleton Pattern: Motivation

Consider two problems:

1. ***Bank accounts*** share a set of data.

   e.g., interest and exchange rates, minimum and maximum balance, *etc*.

2. ***Processes*** are regulated to access some shared, limited resources.

   e.g., printers

Shared Data via Inheritance

Descendant:

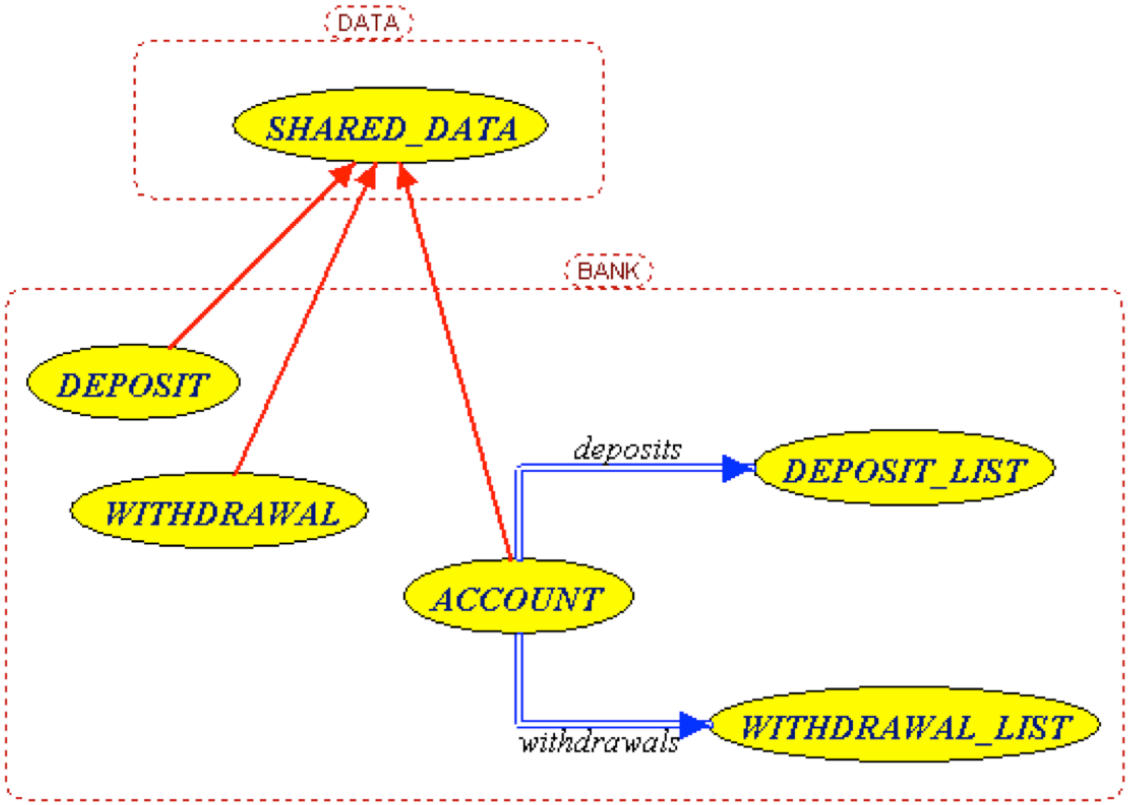
```
class DEPOSIT inherit SHARED_DATA
      -- 'maximum_balance' relevant
end

class WITHDRAW inherit SHARED_DATA
      -- 'minimum_balance' relevant
end

class INT_TRANSFER inherit SHARED_DATA
      -- 'exchange_rate' relevant
end

class ACCOUNT inherit SHARED_DATA
feature
      -- 'interest_rate' relevant
      deposits: DEPOSIT_LIST
      withdraws: WITHDRAW_LIST
end
```

Ancestor:

```
class
  SHARED_DATA
feature
  interest_rate: REAL
  exchange_rate: REAL
  minimum_balance: INTEGER
  maximum_balance: INTEGER
  ...
end
```

Problems?

Sharing Data via Inheritance: Architecture



- *Irreverent* features are inherited.
  ⇒ Descendants' *cohesion* is broken.
- Same set of data is *duplicated* as instances are created.
  ⇒ Updates on these data may result in *inconsistency*.

Sharing Data via Inheritance: Limitation

- Each descendant instance at runtime owns a separate copy of the shared data.
- This makes inheritance *not* an appropriate solution for both problems:
  - What if the interest rate changes? Apply the change to all instantiated account objects?
  - An update to the global lock must be observable by all regulated processes.

  Solution:
  - Separate notions of ***data*** and its ***shared access*** in two separate classes.
  - *Encapsulate* the shared access itself in a separate class.

Introducing the Once Routine in Eiffel (1.1)

```
1  class A
2  create make
3  feature -- Constructor
4    make do end
5  feature -- Query
6    new_once_array (s: STRING): ARRAY[STRING]
7        -- A once query that returns an array.
8      once
9        create {ARRAY[STRING]} Result.make_empty
10       Result.force (s, Result.count + 1)
11     end
12   new_array (s: STRING): ARRAY[STRING]
13       -- An ordinary query that returns an array.
14     do
15       create {ARRAY[STRING]} Result.make_empty
16       Result.force (s, Result.count + 1)
17     end
18 end
```

L9 & L10 executed **only once** for initialization.

L15 & L16 executed **whenever** the feature is called.

Introducing the Once Routine in Eiffel (1.2)

```
1  test_query: BOOLEAN
2   local
3    a: A
4    arr1, arr2: ARRAY[STRING]
5   do
6    create a.make
7
8    arr1 := a.new_array ("Alan")
9    Result := arr1.count = 1 and arr1[1] ~ "Alan"
10   check Result end
11
12   arr2 := a.new_array ("Mark")
13   Result := arr2.count = 1 and arr2[1] ~ "Mark"
14   check Result end
15
16   Result := not (arr1 = arr2)
17   check Result end
18  end
```

Introducing the Once Routine in Eiffel (1.3)

```
1  test_once_query: BOOLEAN
2    local
3      a: A
4      arr1, arr2: ARRAY[STRING]
5    do
6      create a.make
7
8      arr1 := a.new_once_array ("Alan")
9      Result := arr1.count = 1 and arr1[1] ~ "Alan"
10     check Result end
11
12     arr2 := a.new_once_array ("Mark")
13     Result := arr2.count = 1 and arr2[1] ~ "Alan"
14     check Result end
15
16     Result := arr1 = arr2
17     check Result end
18 end
```

Introducing the Once Routine in Eiffel (2)

```
r (...): T
        once
                -- Some computations on Result
                ...
        end
```

- The ordinary **do** ... **end** is replaced by **once** ... **end**.
- The first time the **once** routine *r* is called by some client, it executes the body of computations and returns the computed result.
- From then on, the computed result is "*cached*".
- In every subsequent call to *r*, possibly by different clients, the body of *r* is not executed at all; instead, it just returns the "*cached*" result, which was computed in the very first call.
- **How does this help us?**
  Cache the reference to the same shared object !

Approximating Once Routine in Java (1)

We may encode Eiffel once routines in Java:

```
class BankData {
  BankData() { }
  double interestRate;
  void setIR(double r);
  ...
}
```

```
class Account {
  BankData data;
  Account() {
    data = BankDataAccess.getData();
  }
}
```

```
class BankDataAccess {
  static boolean initOnce;
  static BankData data;
  static BankData getData() {
    if(!initOnce) {
      data = new BankData();
      initOnce = true;
    }
    return data;
  }
}
```

Problem?

Multiple ***BankData*** objects may be created in `Account`, breaking the singleton!

```
Account() {
  data = new BankData();
}
```

Approximating Once Routine in Java (2)

We may encode Eiffel once routines in Java:

```
class BankData {
  private BankData() { }
  double interestRate;
  void setIR(double r);
  static boolean initOnce;
  static BankData data;
  static BankData getData() {
    if(!initOnce) {
      data = new BankData();
      initOnce = true;
    }
    return data;
  }
}
```

Problem?

Loss of Cohesion: ***Data*** and ***Access to Data*** are two separate concerns, so should be decoupled into two different classes!

Singleton Pattern in Eiffel (1)

Supplier:

```
class DATA
create {DATA_ACCESS} make
feature {DATA_ACCESS}
  make do v := 10 end
feature -- Data Attributes
  v: INTEGER
  change_v (nv: INTEGER)
    do v := nv end
end
```

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

Client:

```
test: BOOLEAN
  local
    access: DATA_ACCESS
    d1, d2: DATA
  do
    d1 := access.data
    d2 := access.data
    Result := d1 = d2
      and d1.v = 10 and d2.v = 10
    check Result end
    d1.change_v (15)
    Result := d1 = d2
      and d1.v = 15 and d2.v = 15
  end
end
```

Writing **create** *d1.make* in `test` feature does not compile. Why?

Singleton Pattern in Eiffel (2)

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
```

Client:

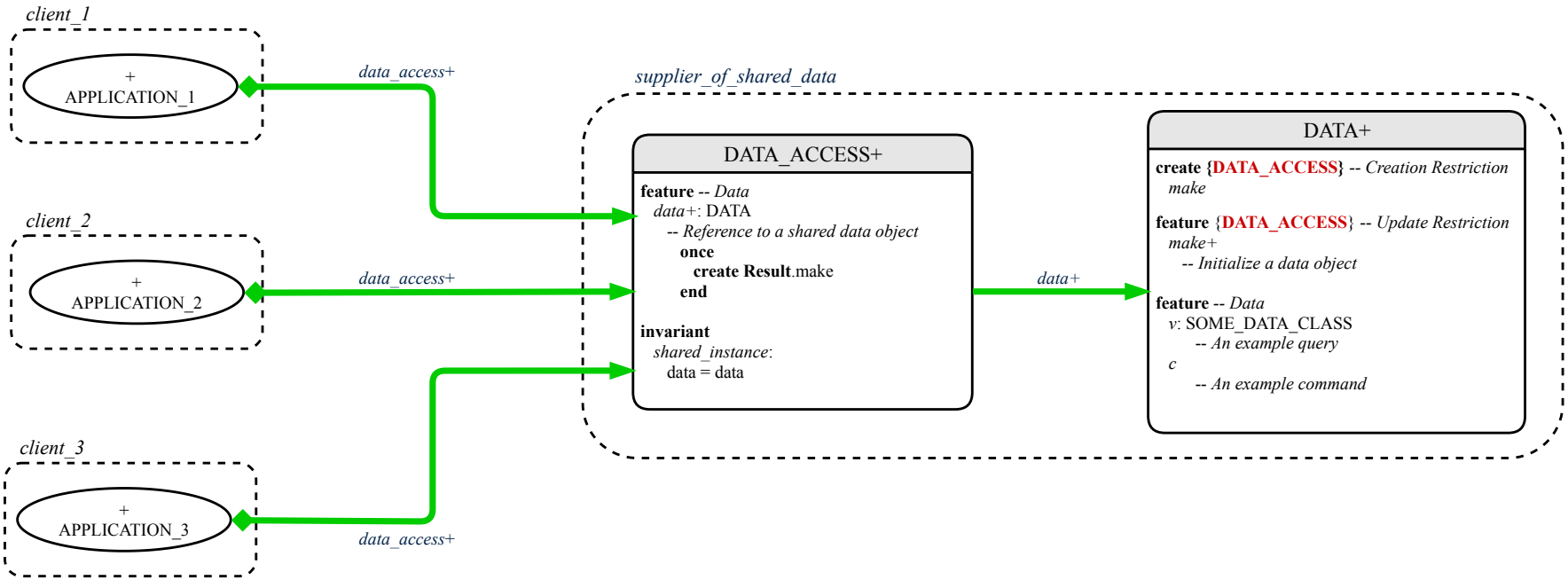
```
class
  ACCOUNT
feature
  data: BANK_DATA
  make (...)
      -- Init. access to bank data.
    local
      data_access: BANK_DATA_ACCESS
    do
      data := data_access.data
      ...
    end
end
```

Writing **create** *data.make* in client's `make` feature does not compile. Why?

Testing Singleton Pattern in Eiffel

```
test_bank_shared_data: BOOLEAN
    -- Test that a single data object is manipulated
  local acc1, acc2: ACCOUNT
  do
    comment("t1: test that a single data object is shared")
    create acc1.make ("Bill")
    create acc2.make ("Steve")
    Result := acc1.data = acc2.data
    check Result end
    Result := acc1.data ~ acc2.data
    check Result end
    acc1.data.set_interest_rate (3.11)
    Result :=
          acc1.data.interest_rate = acc2.data.interest_rate
      and acc1.data.interest_rate = 3.11
    check Result end
    acc2.data.set_interest_rate (2.98)
    Result :=
          acc1.data.interest_rate = acc2.data.interest_rate
      and acc1.data.interest_rate = 2.98
  end
```

Singleton Pattern: Architecture



Important Exercises: Instantiate this architecture to the problem of shared bank data.

Draw it in `draw.io`.

Beyond this lecture

The ***singleton*** pattern is instantiated in the ETF framework:

- `ETF_MODEL` (***shared data***)
- `ETF_MODEL_ACCESS` (***exclusive once access***)
- `ETF_COMMAND` and its effective descendants:

```
deferred class
  ETF_COMMAND
feature -- Attributes
  model: ETF_MODEL
feature {NONE}
  make(...)
    local
      ma: ETF_MODEL_ACCESS
    do
      ...
      model := ma.m
    end
end
```

```
class
  ETF_MOVE
inherit
  ETF_MOVE_INTERFACE
  -- which inherits ETF_COMMAND
feature -- command
  move(...)
    do
      ...
      model.some_routine (...)
      ...
    end
end
```

Index (1)

Index (2)