Lecture 13

Monday, February 24

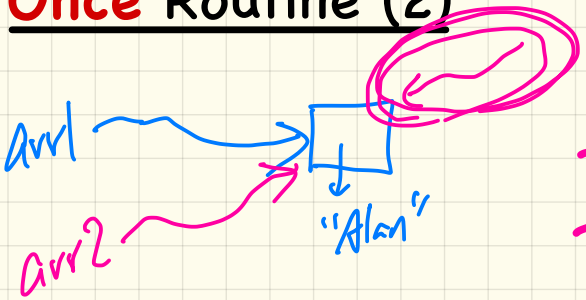
- Office hours today moved to
  Tuesday 12:30 - 14:30
- Moodle announcement on Labtest1
- ETF Tutorial
  Video 7 on automating acceptance testing

Once Routine (1)

```
class A
create make
feature -- Constructor
  make do end
feature -- Query
  new_once_array (s: STRING): ARRAY[STRING]
      -- A once query that returns an array.
    once
      create {ARRAY[STRING]} Result.make_empty
      Result.force (s, Result.count + 1)
    end
  new_array (s: STRING): ARRAY[STRING]
      -- An ordinary query that returns an array.
    do
      create {ARRAY[STRING]} Result.make_empty
      Result.force (s, Result.count + 1)
    end
end
```

```
test_query: BOOLEAN
  local
    a: A
    arr1, arr2: ARRAY[STRING]
  do
    create a.make

    arr1 := a.new_array ("Alan")
    Result := arr1.count = 1 and arr1[1] ~ "Alan"
    check Result end

    arr2 := a.new_array ("Mark")
    Result := arr2.count = 1 and arr2[1] ~ "Mark"
    check Result end

    Result := not (arr1 = arr2)
    check Result end
  end
```

Once Routine (2)

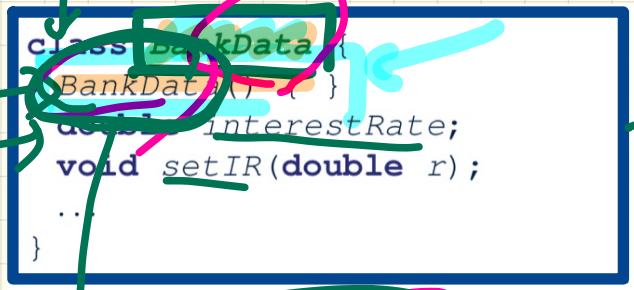


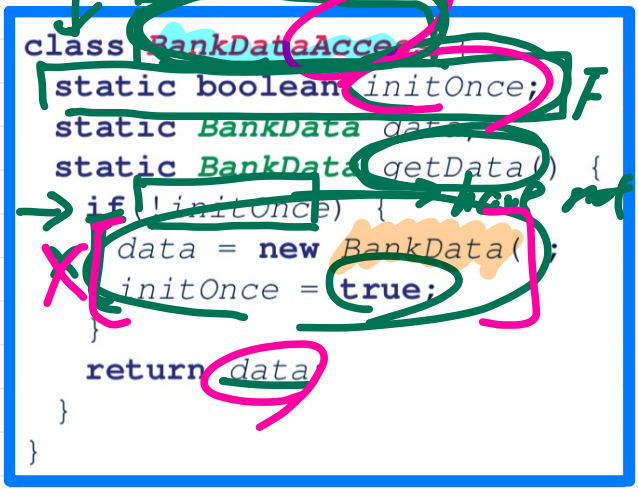

```
class A
create make
feature -- Constructor
  make do end
feature -- Query
  new_once_array (s: STRING): ARRAY[STRING]
      -- A once query that returns an array.
    once
      create {ARRAY[STRING]} Result.make_empty
      Result.force (s, Result.count + 1)
    end
  new_array (s: STRING): ARRAY[STRING]
      -- An ordinary query that returns an array.
    do
      create {ARRAY[STRING]} Result.make_empty
      Result.force (s, Result.count + 1)
    end
end
```

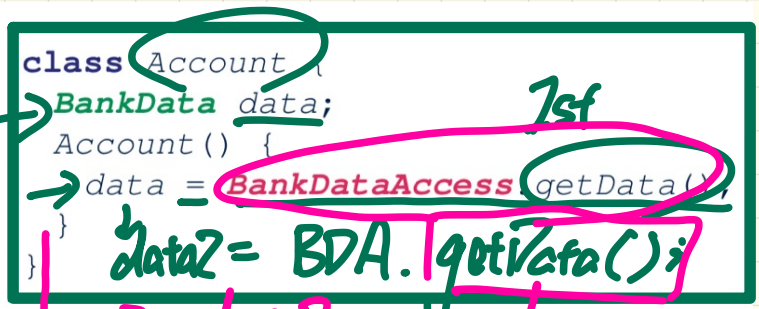
```
test_once_query: BOOLEAN
  local
    a: A
    arr1, arr2: ARRAY[STRING]
  do
    create a.make
    arr1 := a.new_once_array ("Alan")
    Result := arr1.count = 1 and arr1[1] ~ "Alan"
    check Result end

    arr2 := a.new_once_array ("Mark")
    Result := arr2.count = 1 and arr2[1] ~ "Alan"
    check Result end

    Result := arr1 = arr2
    check Result end
end
```

- Cohesion
- Single chore principle
- Single data instance

Approximating Once Routines in Java (1)







Problem?

Approximating Once Routines 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;
  }
}
```

data

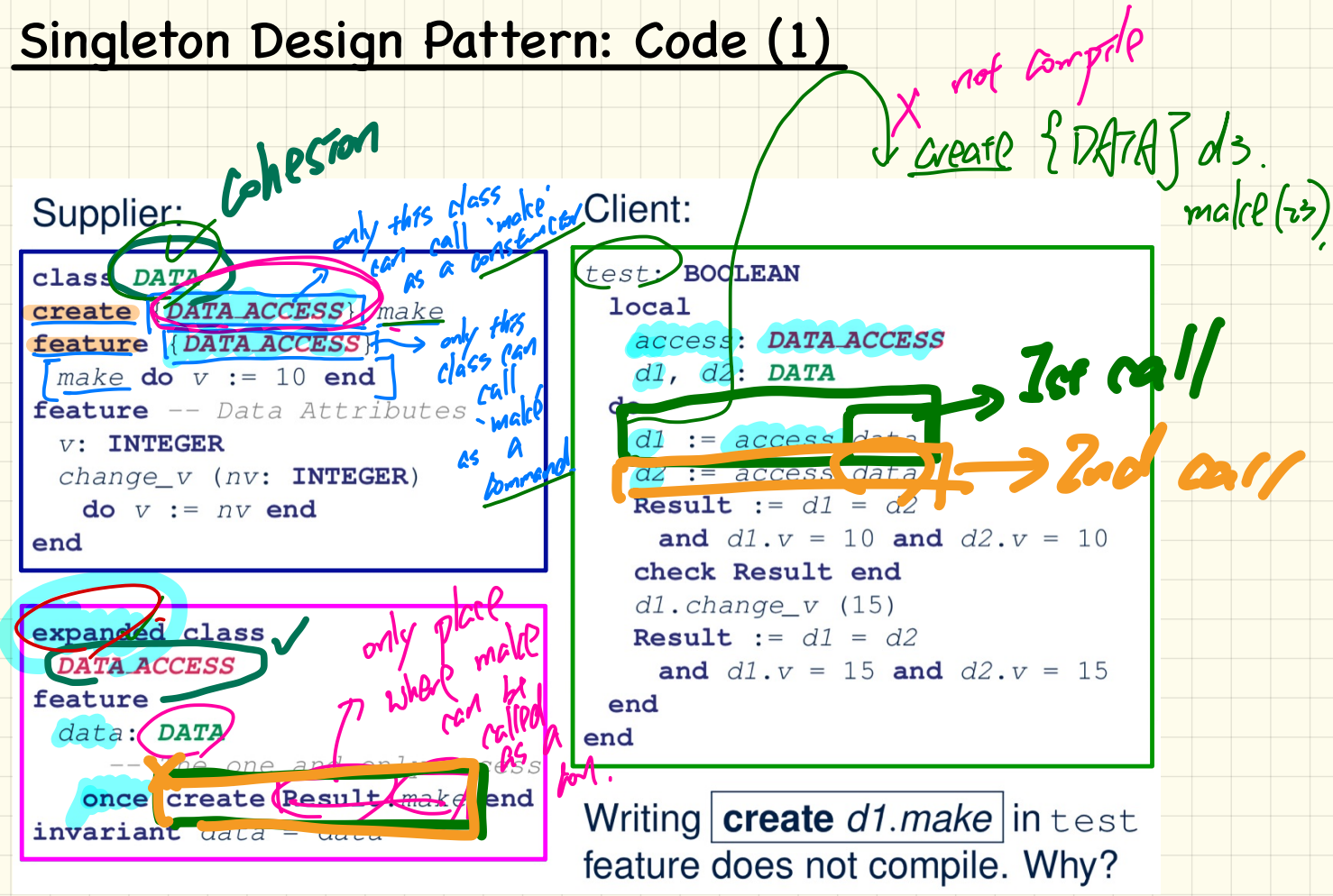
data access

Problem?

Teste

BD d = new BD(); X

Singleton Design Pattern: Code (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?

Supplier:

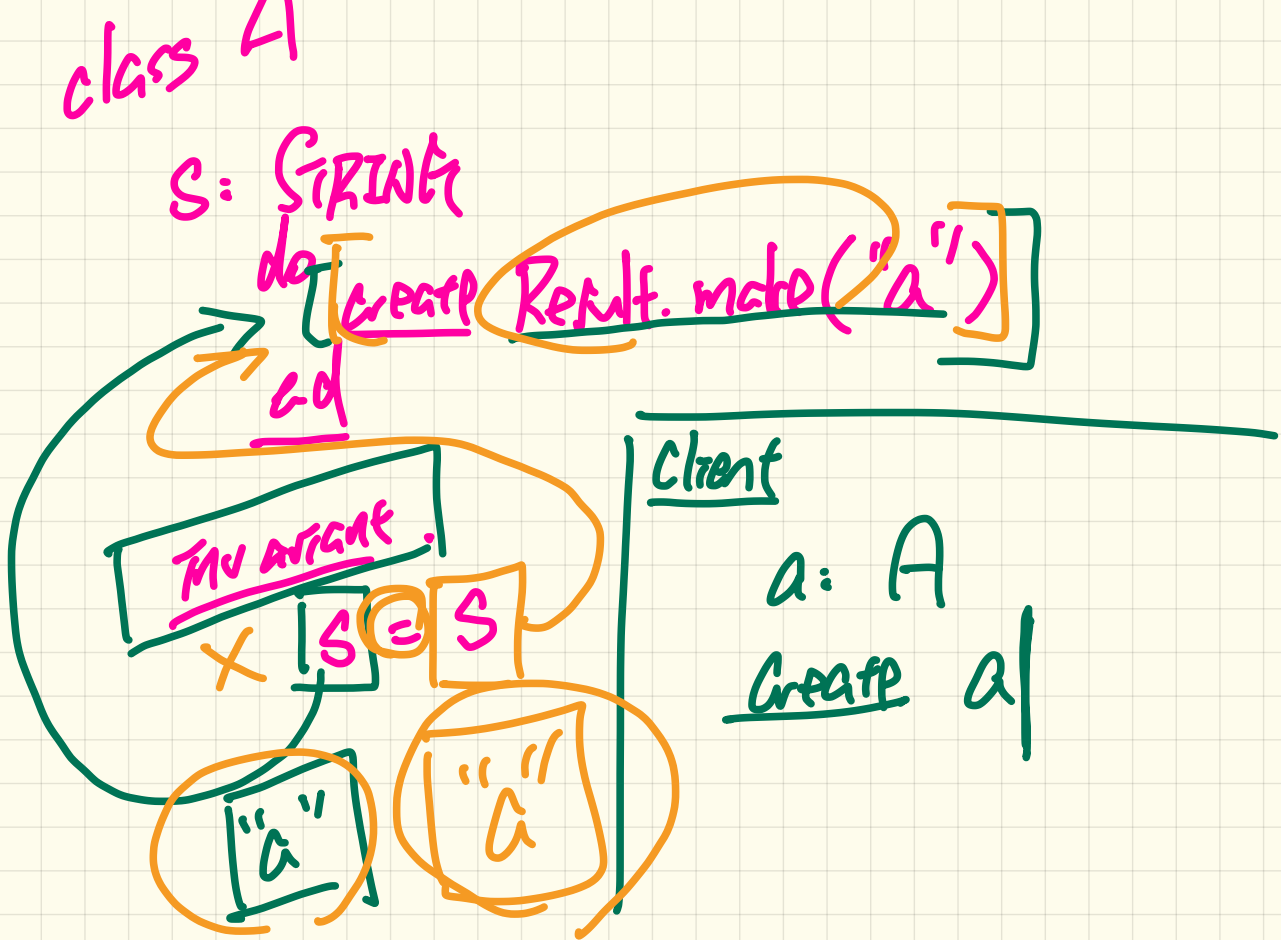
```
class DATA
create {DATA_ACCESS} make
feature {DATA_ACCESS}
  make do v := 10 end
feature -- Data Attributes
  v: INTEGER
  change_v (nv: INTEGER)
    do v := nv end
end
```

```
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
```

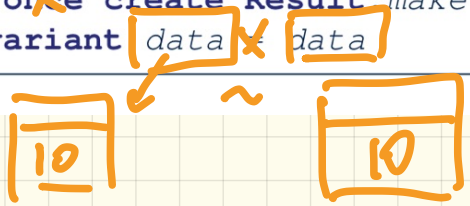
Writing **create** *d1.make* in `test` feature does not compile. Why?


class A
s: STRING
do
create Result.make("a")
end
Invariant
s = s
"a"
"a"
Client
a: A
create a

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?

Export Status Case 1

```
class CLIENT_1
  ...
  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 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
    make

feature
  make (init_i: INTEGER)
    do
      i := init_i
    end

feature
  i: INTEGER
end
```

Export Status Case 2

```
class CLIENT_1
  ...
  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 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
    make

feature
    make (init_i: INTEGER)
        do
            i := init_i
        end

feature
    i: INTEGER
end
```

Singleton Design Pattern: Code (2.1)

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
```

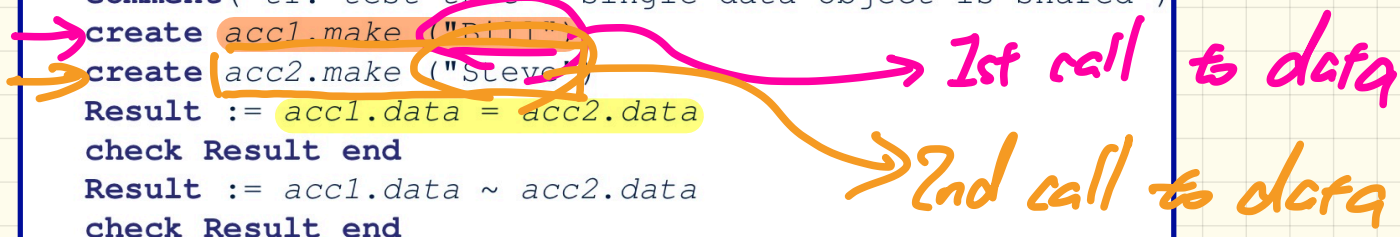
Client:

```
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?

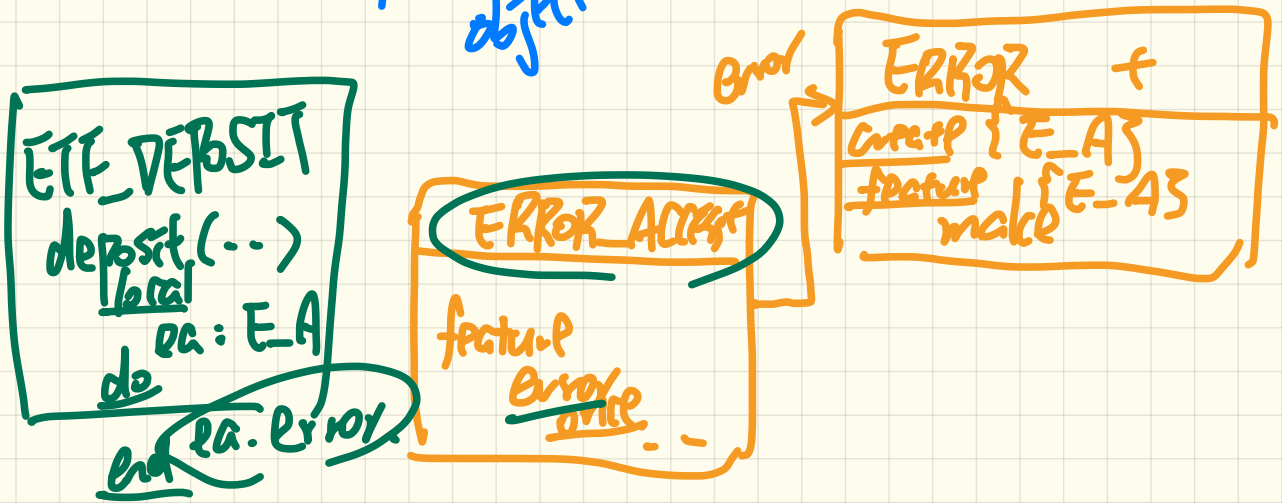
Singleton Design Pattern: Code (2.2)

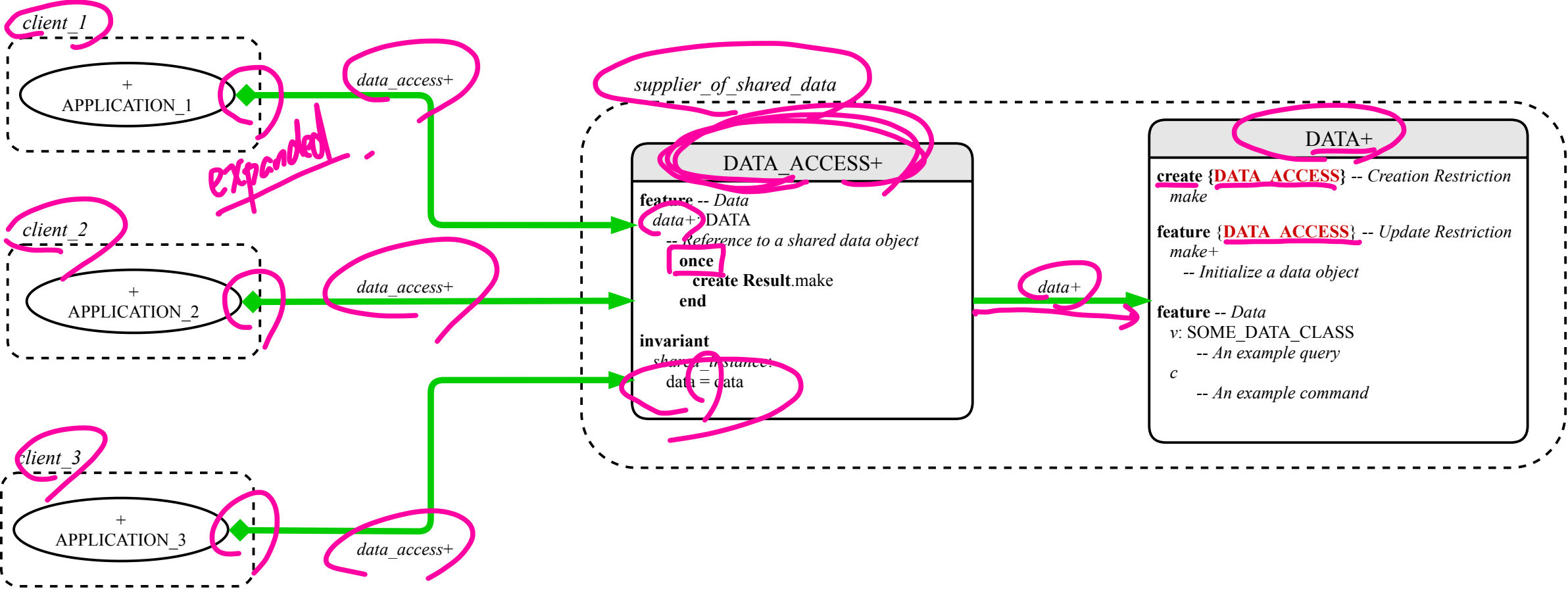
```
test_bank_shared_data: BOOLEAN
    -- Test that a single data object is manipulated
  local acc1, acc2: ACCOUNT
  do
    comment("t1: test that a single data object is shared")
    create acc1.make ("Bill")
    create acc2.make ("Steve")
    Result := acc1.data = acc2.data
    check Result end
 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



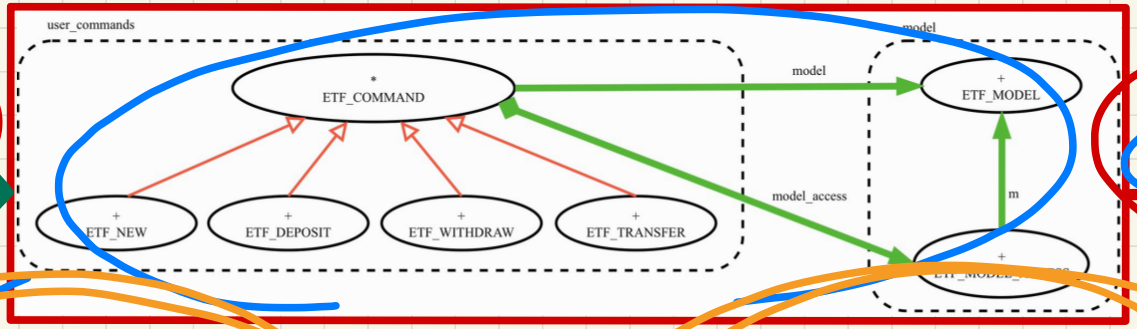
Error Handling using Singleton

- only a single shared error object at runtime.




client_1
+
APPLICATION_1
data_access+
expanded
client_2
+
APPLICATION_2
data_access+
client_3
+
APPLICATION_3
data_access+
supplier_of_shared_data
DATA_ACCESS+
feature -- Data
data+: DATA
-- Reference to a shared data object
once
create Result.make
end
invariant
shared_instance:
data = data
data+
DATA+
create {DATA_ACCESS} -- Creation Restriction
make
feature {DATA_ACCESS} -- Update Restriction
make+
-- Initialize a data object
feature -- Data
v: SOME_DATA_CLASS
-- An example query
c
-- An example command

ETF: Input-Output-Based Acceptance Testing



input

output

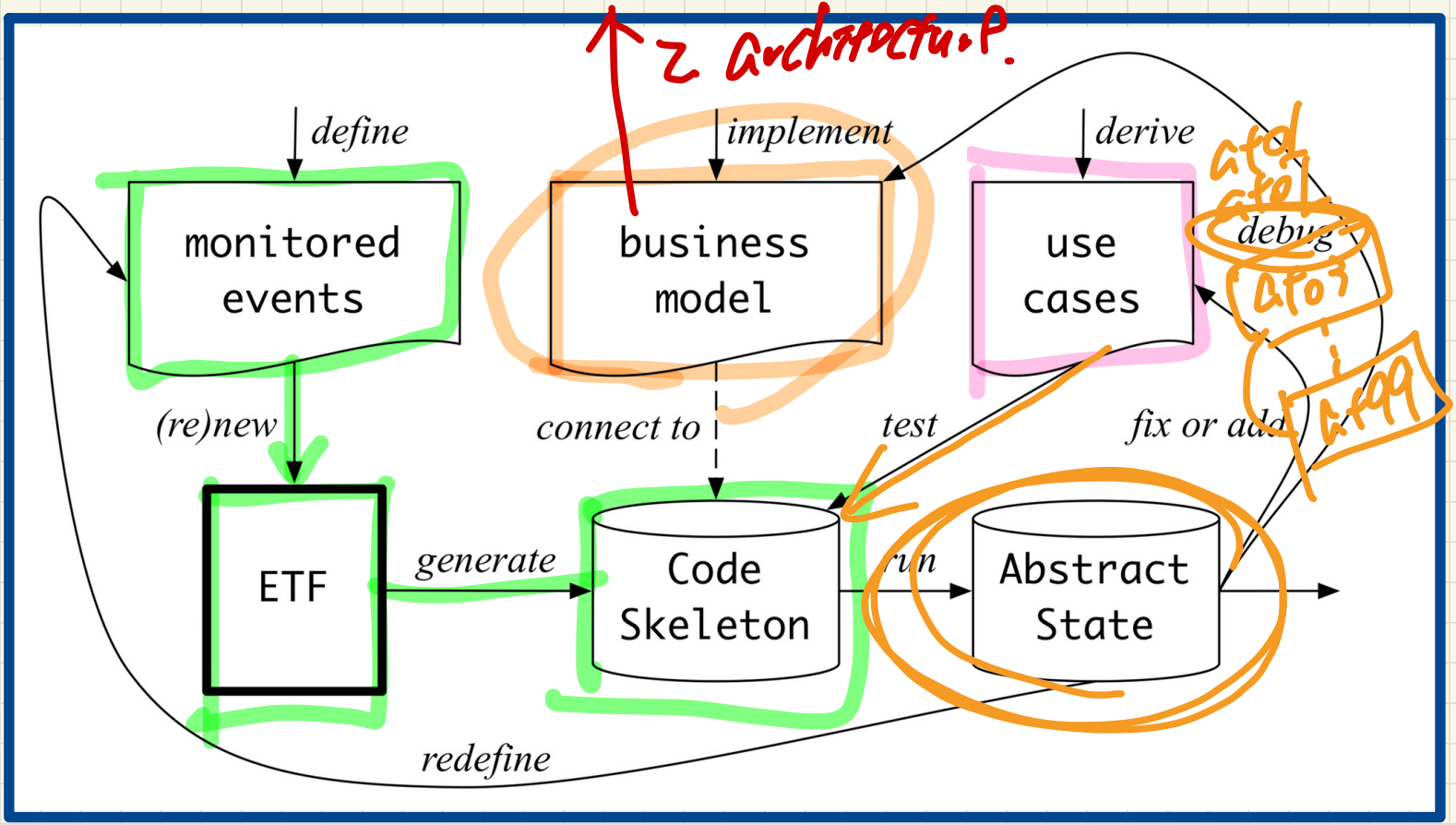
```
new("alan")
new("mark")
deposit("alan", 200)
deposit("mark", 100)
transfer("alan", "mark", 50)
```

```
{}
-> new("alan")
{alan: 0}
-> new("mark")
{alan: 0, mark: 0}
-> deposit("alan", 200)
{alan: 200, mark: 0}
-> deposit("mark", 100)
{alan: 200, mark: 100}
-> transfer("alan", "mark", 50)
{alan: 150, mark: 150}
```

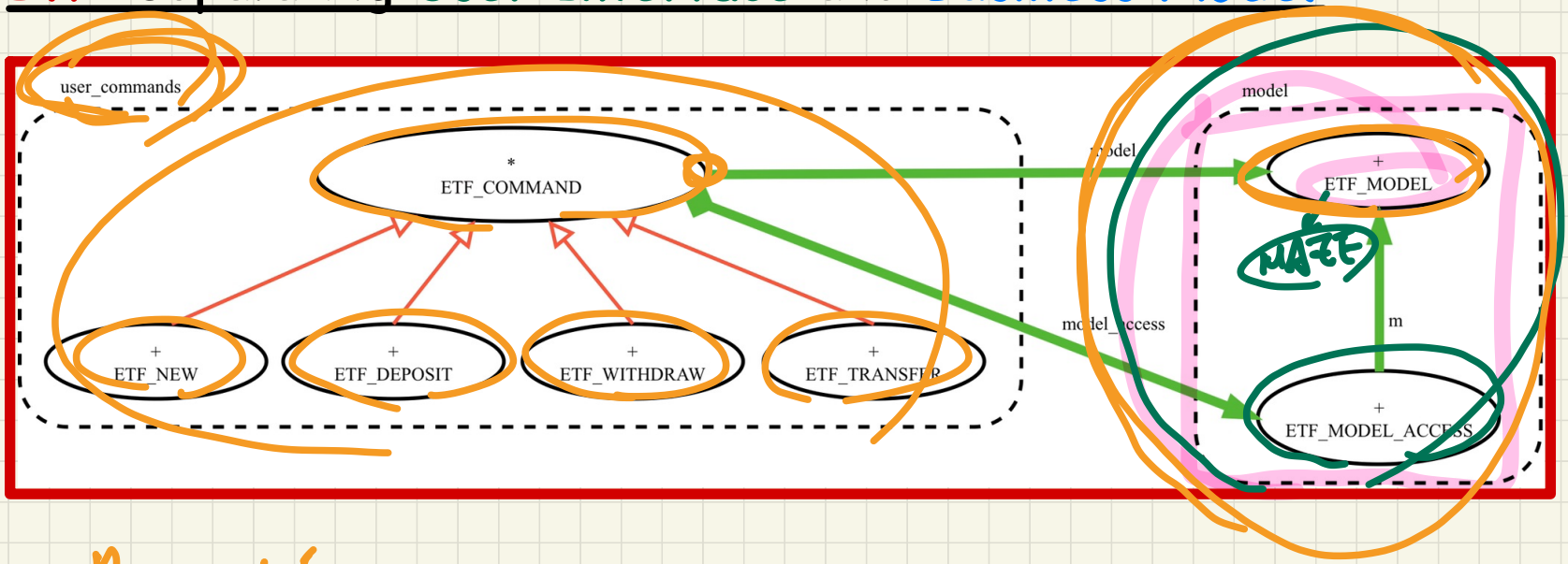
acceptance test (no syntax of a specific prog. lang)

abstract state

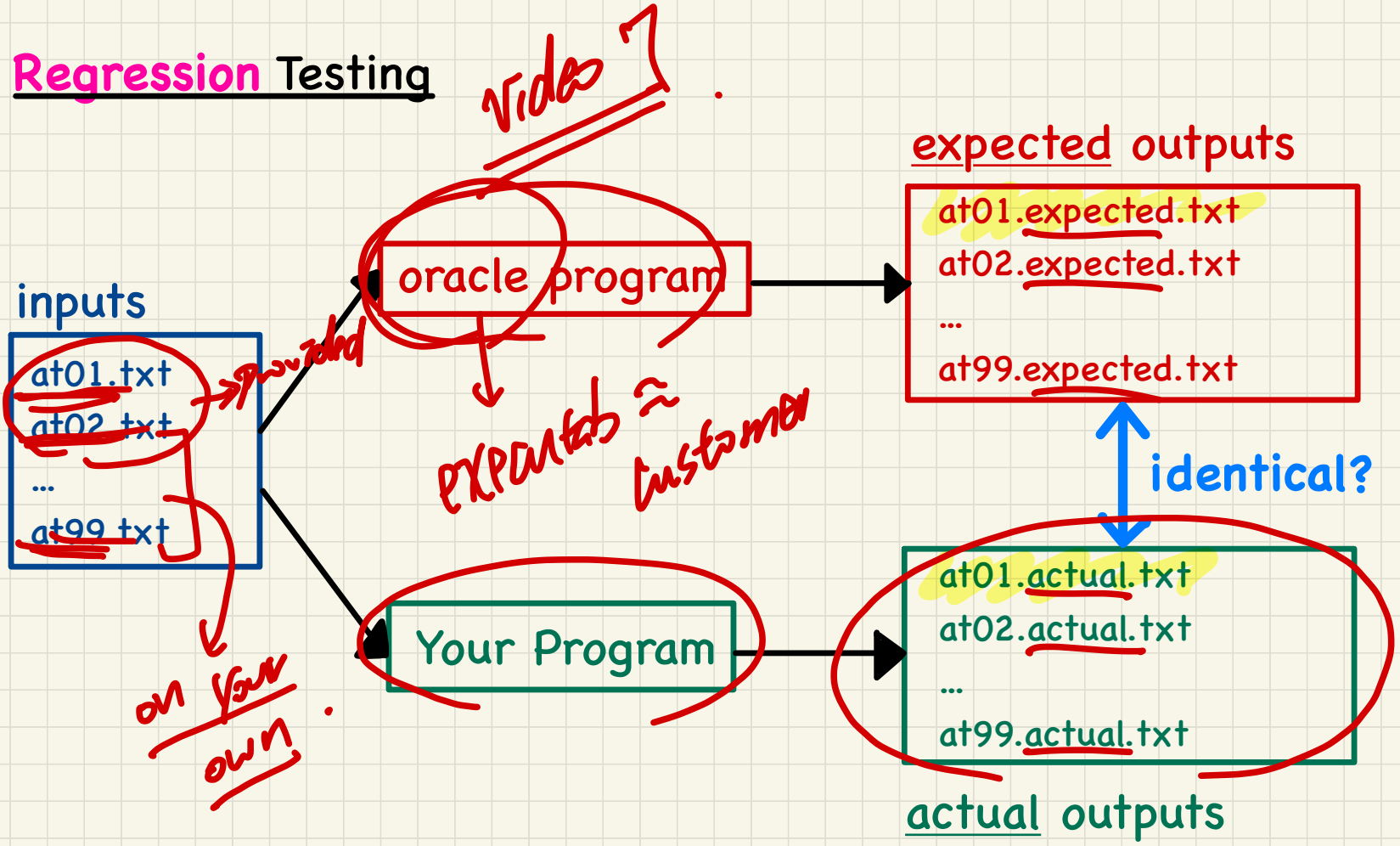
ETF: Workflow



ETF: Separating User Interface and Business Model



Regression Testing



Automating Regression Testing