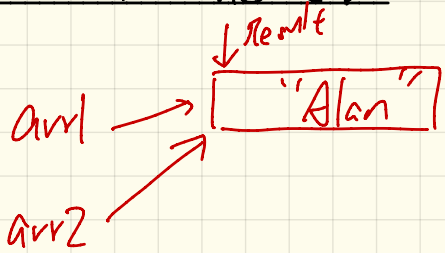
Monday February 4

Lecture 9

- Lab 4 released

Tutorial Videos on ETF

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
```

1st time

not 1st time

ignored.

Approximating Once Routines in Java (1)

Breaking singleton.

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

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

BankData d2 = new BankData();
data == d2

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

false

factory method

Problem?


```
BankDataAccess bda = new ____();
BankData d1 = bda.getData();
BankData d2 = bda.getData();
d1.setIR(1.23);
```

Approximating Once Routines in Java (2)

Separation of Concerns: Data / DataAccess

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;
  }
}
```

static

data

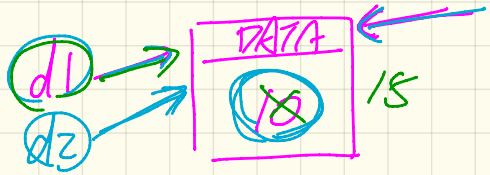
data access

Problem?

```
BankData d1 = new
BankData();
```

✗ :(it's private.

Singleton 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
```

create access.make ; It's expanded

1st call

not 1st call

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

Singleton Pattern: Code (2.1)

class BANK_DATA_ACCESS
data : BD
once create Result.make
end

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
```

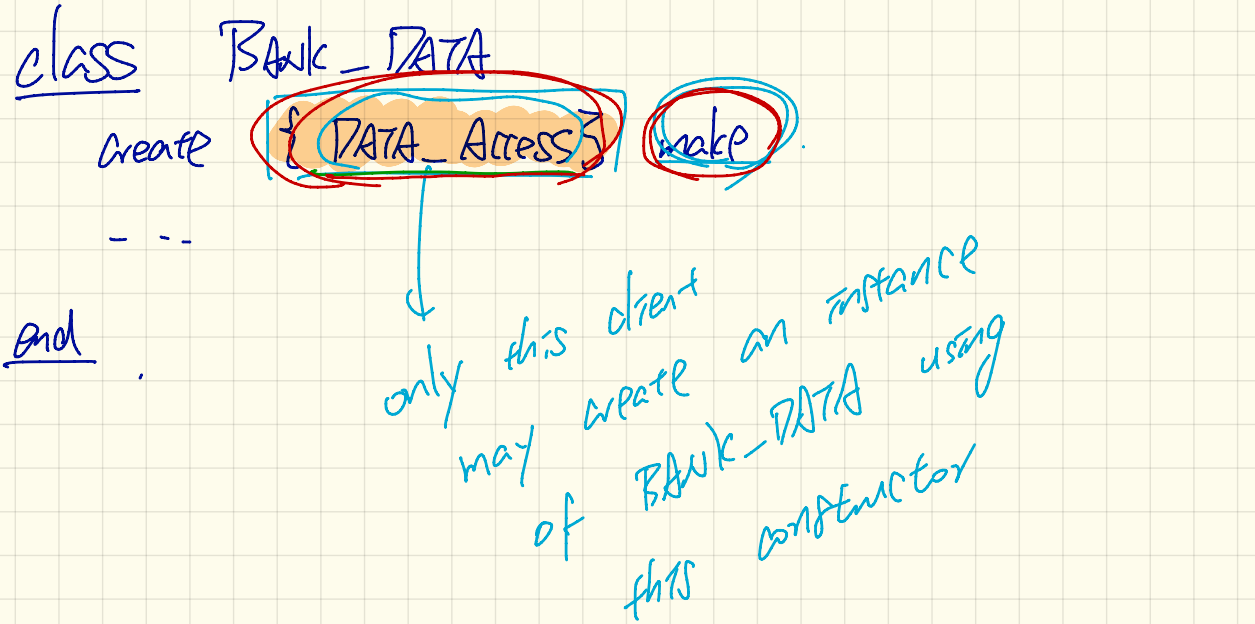
```
expanded class
  BANK_DATA_ACCESS
feature
  data: BANK_DATA
      -- The one and only access
    once create Result.make end
invariant data = data
```

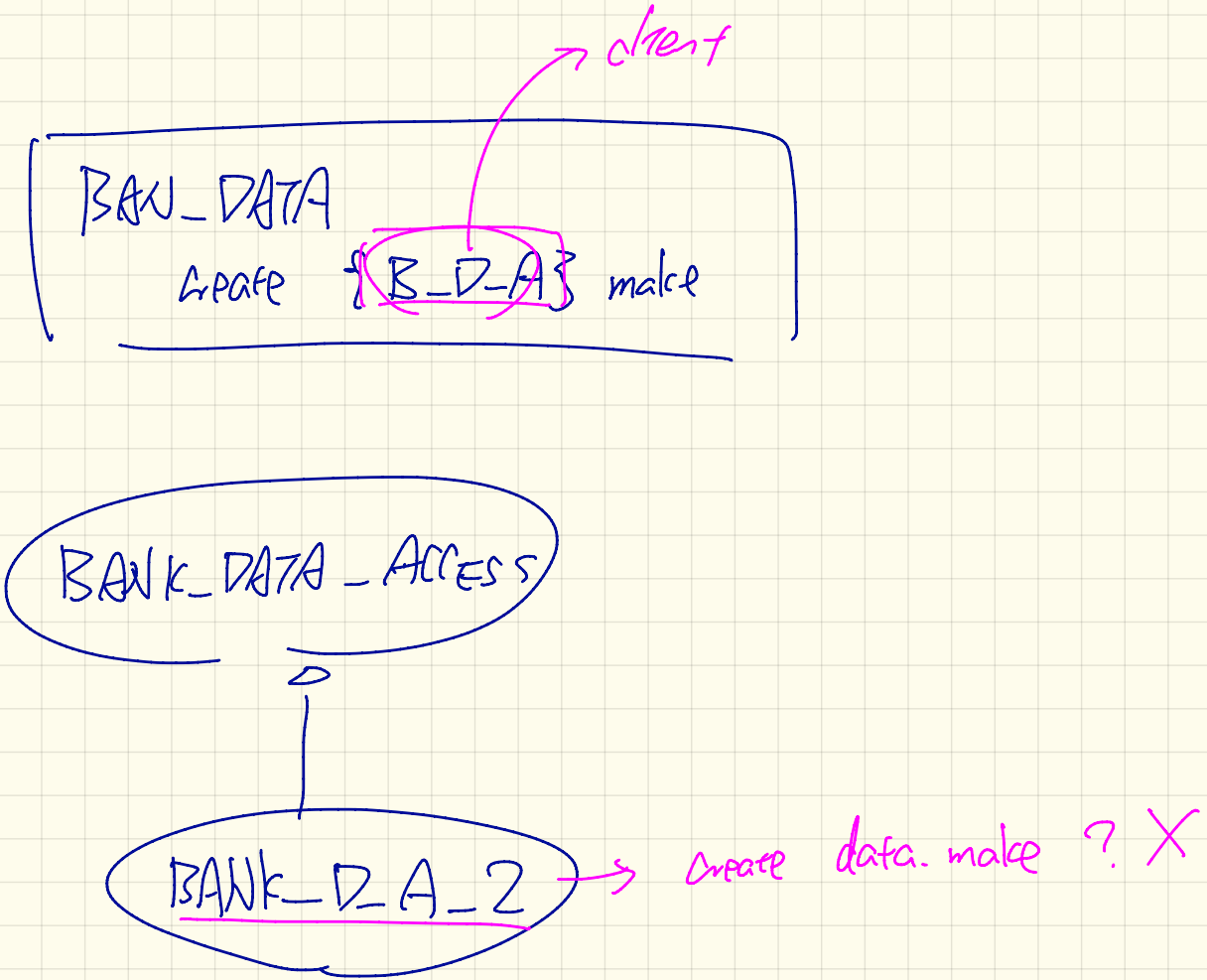
Client:

```
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
```

create data.make

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


class BANK_DATA
create {DATA_Access} make
...
end
only this client may create an instance of BANK_DATA using this constructor
class ACCOUNT
f
local bd: BANK_DATA
do
end create bd.make
end
da: DATA_ACCESS
bd := da.data


client
BAN_DATA
create
B_D_A
make
BANK_DATA_Access
BANK_D_A_2
create
data. make ? X

class ACCOUNT

make

local

da1, da2 : B_D_A

data1, data2 : B_D

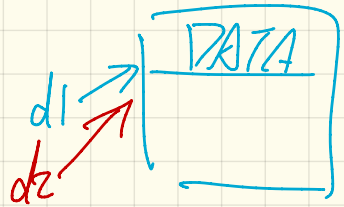
do

→ data1 := da2.data

data2 := da1.data →

end

end



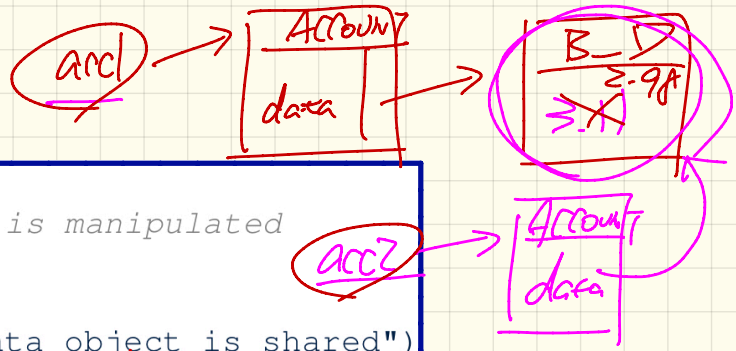
data1 = data2.

2nd time

that {B_D_A}.data

once routine is called

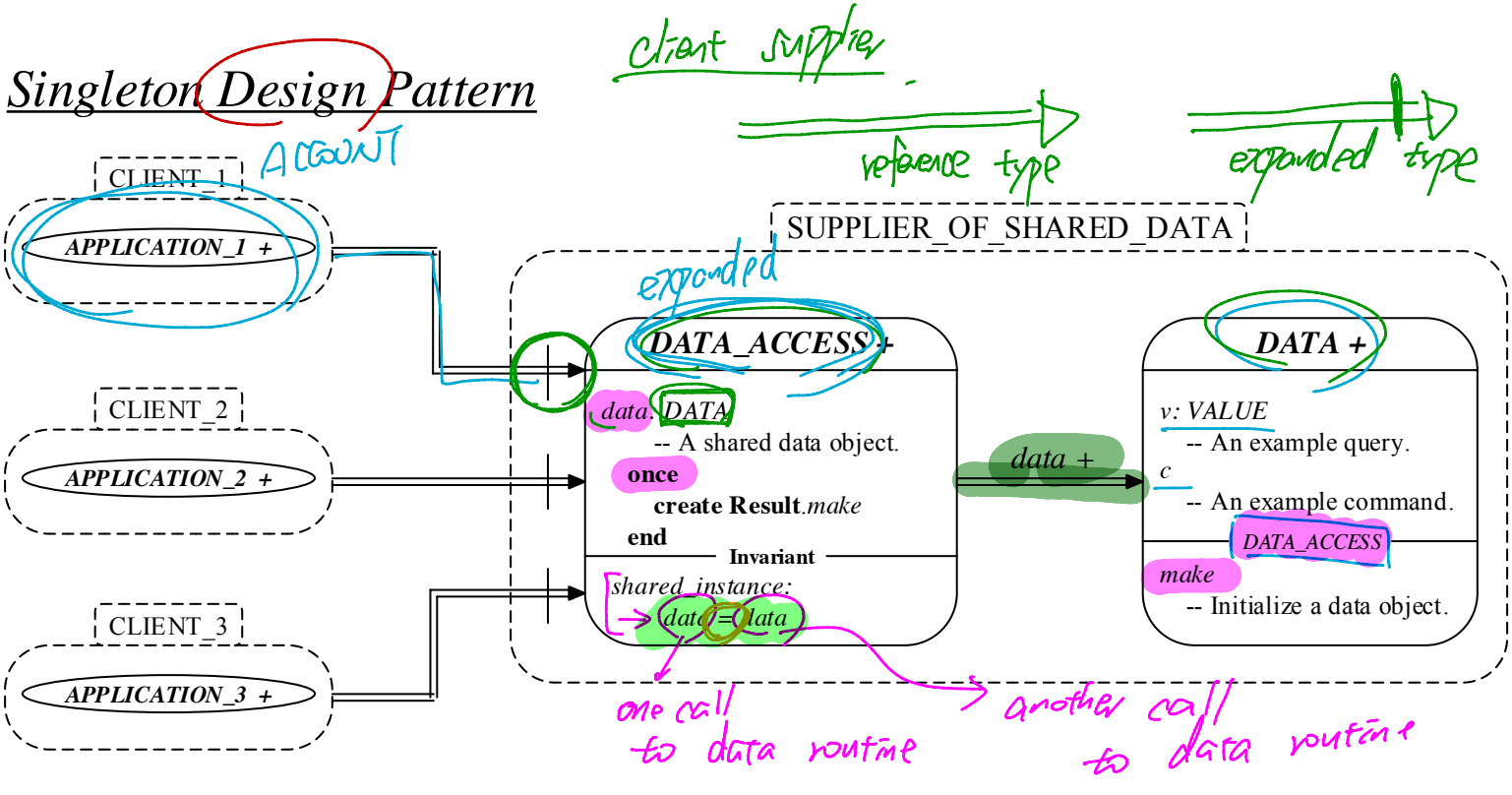
Singleton Pattern: Code (2.2)




```
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
```

1st time calling {B_DATA}.data

Singleton Design Pattern

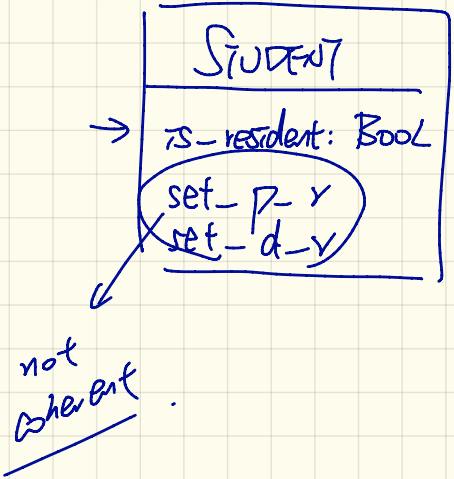


$$= o1 := o2$$

$$= o.f(o2)$$

$$= o1 := f(o1)$$

8 kinds of students.



$b_1, b_2, b_3 : \text{BOOL}$

$b_1 \wedge \neg b_2 \wedge b_3 \longrightarrow$ I kind

$\neg b_1 \wedge b_2 \wedge \neg b_3 \longrightarrow$ another kind.

Violation of Single Choice Principle

```
class NON_RESIDENT_STUDENT
create make
feature -- Attributes
  name: STRING
  courses: LINKED_LIST[COURSE]
  discount_rate: REAL
feature -- Constructor
  make (n: STRING)
    do name := n ; create courses.make end
feature -- Commands
  set_dr (r: REAL) do discount_rate := r end
  register (c: COURSE) do courses.extend (c) end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := 0.0
       across courses as c loop base := base + c.item.fee end
       Result := base * discount_rate * inf_rate
    end
end
```

```
class RESIDENT_STUDENT
create make
feature -- Attributes
  name: STRING
  courses: LINKED_LIST[COURSE]
  premium_rate: REAL
feature -- Constructor
  make (n: STRING)
    do name := n ; create courses.make end
feature -- Commands
  set_pr (r: REAL) do premium_rate := r end
  register (c: COURSE) do courses.extend (c) end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := 0.0
       across courses as c loop base := base + c.item.fee end
       Result := base * premium_rate * inf_rate
    end
end
```

Without Inheritance: Collection of Students

too tolerant

→ students: LL [ANY] ✗

```
class STUDENT_MANAGEMENT_SYSETM
→rs : LINKED_LIST[RESIDENT_STUDENT]
→nrs : LINKED_LIST[NON_RESIDENT_STUDENT]
  add_rs (rs: RESIDENT_STUDENT) do ... end
  add_nrs (nrs: NON_RESIDENT_STUDENT) do ... end
  register_all (Course c) -- Register a common course 'c'.
    do
    →across rs as c loop c.item.register (c) end
    →across nrs as c loop c.item.register (c) end
    end
end
```

polymorphic collection

Inheritance: Code Reuse

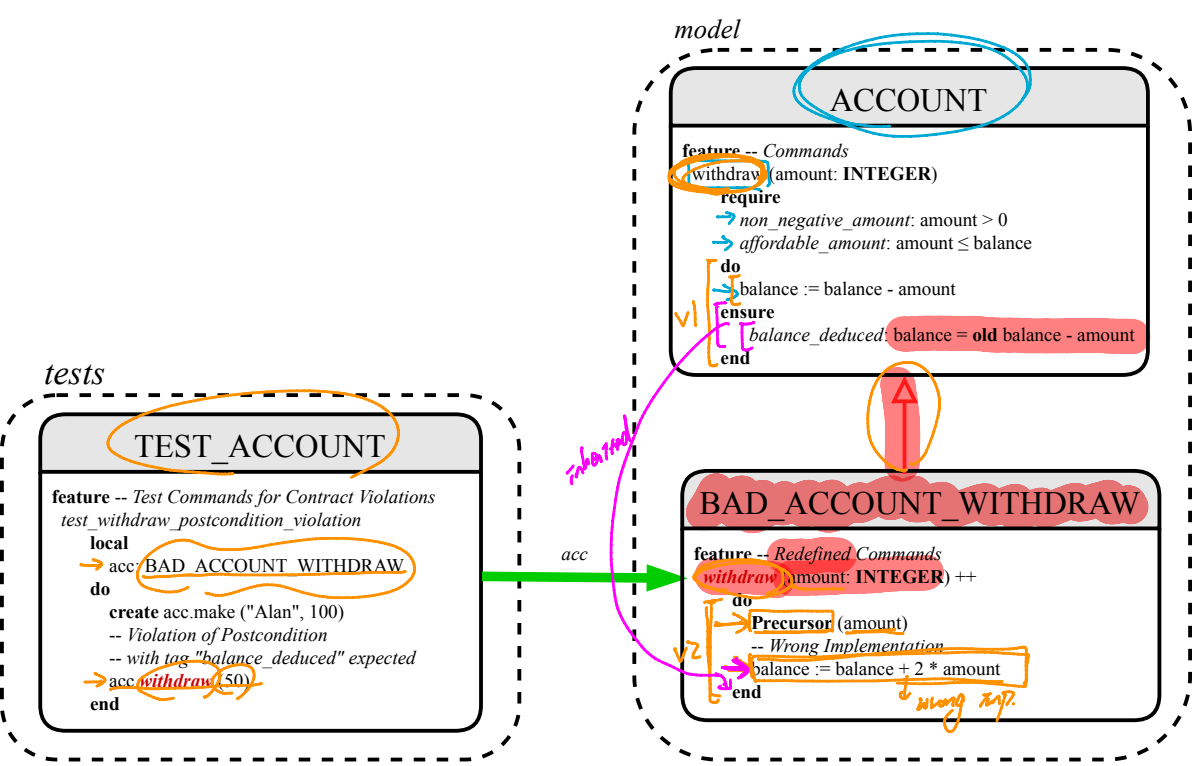
```
class STUDENT
create make
feature -- Attributes
  name: STRING
  courses: LINKED_LIST[COURSE]
feature -- Commands that can be used as constructors.
  make (n: STRING) do name := n ; create courses.make end
feature -- Commands
  register (c: COURSE) do courses.extend (c) end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := 0.0
       across courses as c loop base := base + c.item.fee end
       Result := base
    end
end
```

```
class
  RESIDENT_STUDENT
inherit
  STUDENT
    redefine tuition end
create make
feature -- Attributes
  premium_rate : REAL
feature -- Commands
  set_pr (r: REAL) do premium_rate := r end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := Precursor ; Result := base * premium_rate end
end
```

refers to the version defined in super class

```
class
  NON_RESIDENT_STUDENT
inherit
  STUDENT
    redefine tuition end
create make
feature -- Attributes
  discount_rate : REAL
feature -- Commands
  set_dr (r: REAL) do discount_rate := r end
feature -- Queries
  tuition: REAL
    local base: REAL
    do base := Precursor ; Result := base * discount_rate end
end
```

Precursor(...) →


model
ACCOUNT
feature -- Commands
withdraw (amount: INTEGER)
require
non_negative_amount: amount > 0
affordable_amount: amount ≤ balance
do
balance := balance - amount
ensure
balance_deduced: balance = old balance - amount
end
v1
inherit
BAD_ACCOUNT_WITHDRAW
feature -- Redefined Commands
withdraw (amount: INTEGER) ++
do
Precursor (amount)
-- Wrong Implementation
balance := balance + 2 * amount
end
v2
wrong imp.
acc
tests
TEST_ACCOUNT
feature -- Test Commands for Contract Violations
test_withdraw_postcondition_violation
local
acc: BAD_ACCOUNT_WITHDRAW
do
create acc.make ("Alan", 100)
-- Violation of Postcondition
-- with tag "balance_deduced" expected
acc.withdraw (50)
end