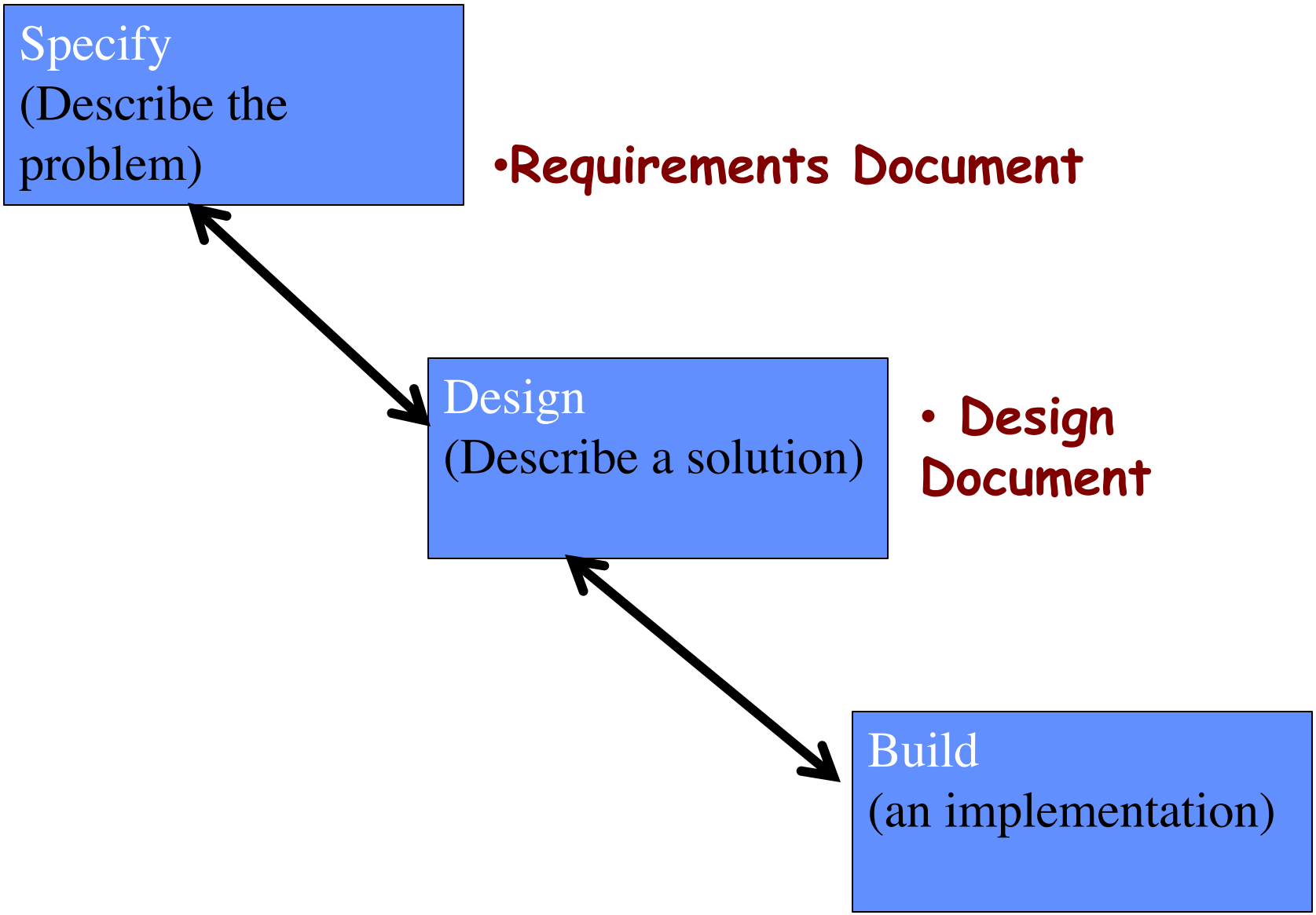
YORK
UNIVERSITÉ
UNIVERSITY

Software Design
CSE3311 Winter 2019

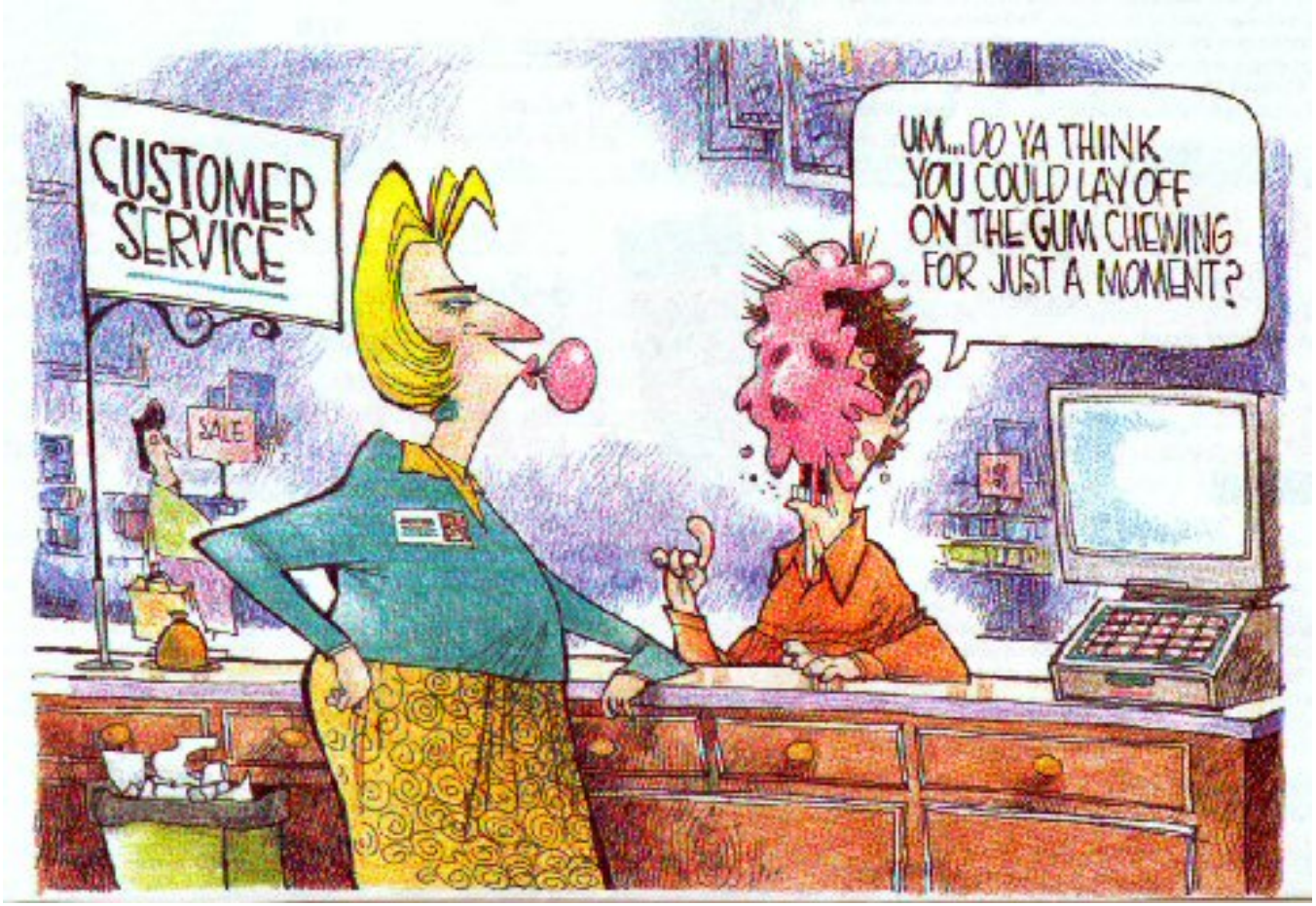
https://wiki.cse.yorku.ca/course_archive/2018-19/W/3311

Design

- Design by Contract
- Unit Testing/Test Driven Design
- BON/UML diagrams
- Object vs. Reference comparison
- Information hiding (LIST[G], and generic parameters)
- Code: bank_account3


Specify
(Describe the problem)
•Requirements Document
Design
(Describe a solution)
• Design Document
Build
(an implementation)

Pleasing your customer



If your customer is unhappy then everybody is unhappy!

"Thank you for calling Customer Service.
If you're calm and rational, press 1.
If you're a whiner, press 2.
If you're a hot head, press 3...."

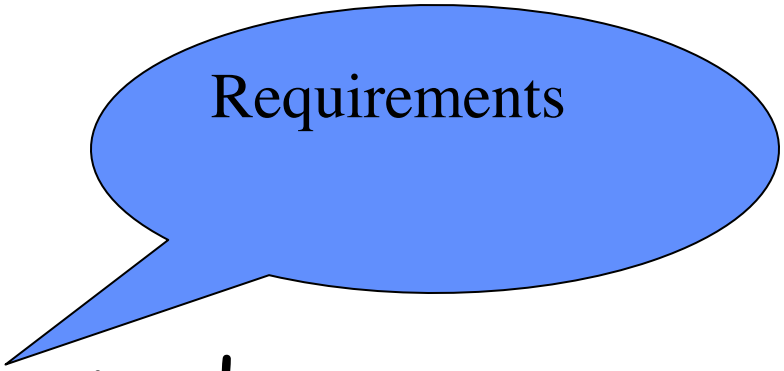
What are the Goals of Good Design?

- ?
- ?
- ?

What is Design?

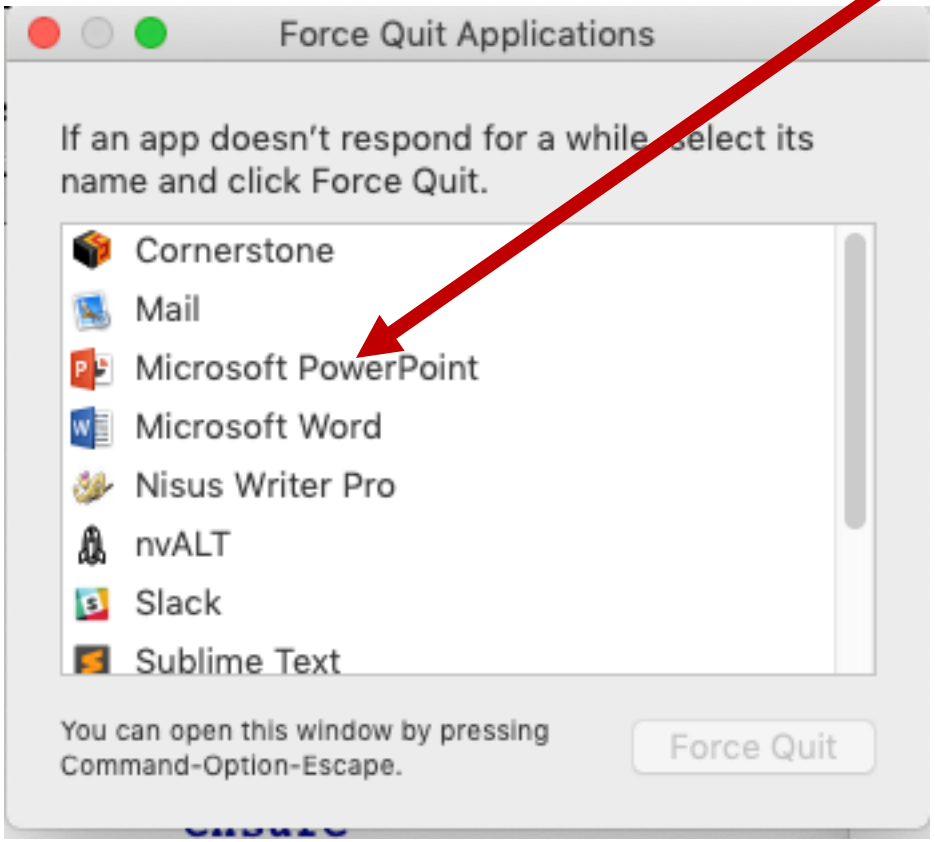
- ?
- ?

Pleasing your customer



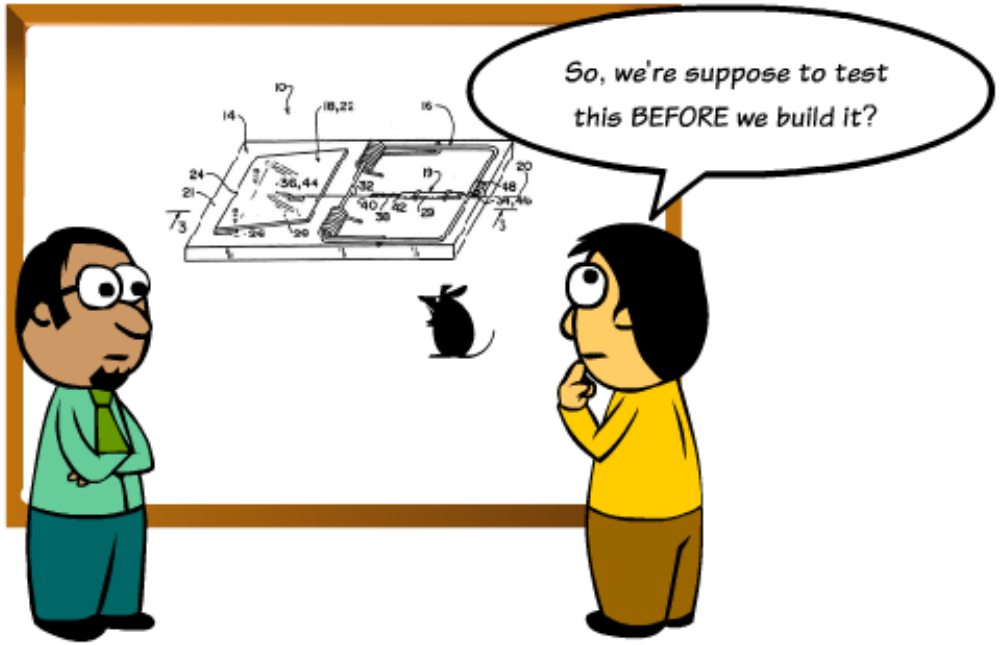
- Find out What the customer needs
- Deliver On Time/Within Budget

This crashed on me today



Bank User Requirements

- **R1: Account balances always exceed the credit limit**
- **R2: Clients can deposit and withdraw dollars**
- R3: Tellers (but not clients) can date a dollar deposit or withdrawal differently than today (e.g. as "tomorrow")
- R4: Tellers (but not clients) can access withdrawals on a given date
- R5: Maximum total withdrawal per day is $5000
- R6: Clients can access the total amount of dollars deposited and dollars withdrawn from their account

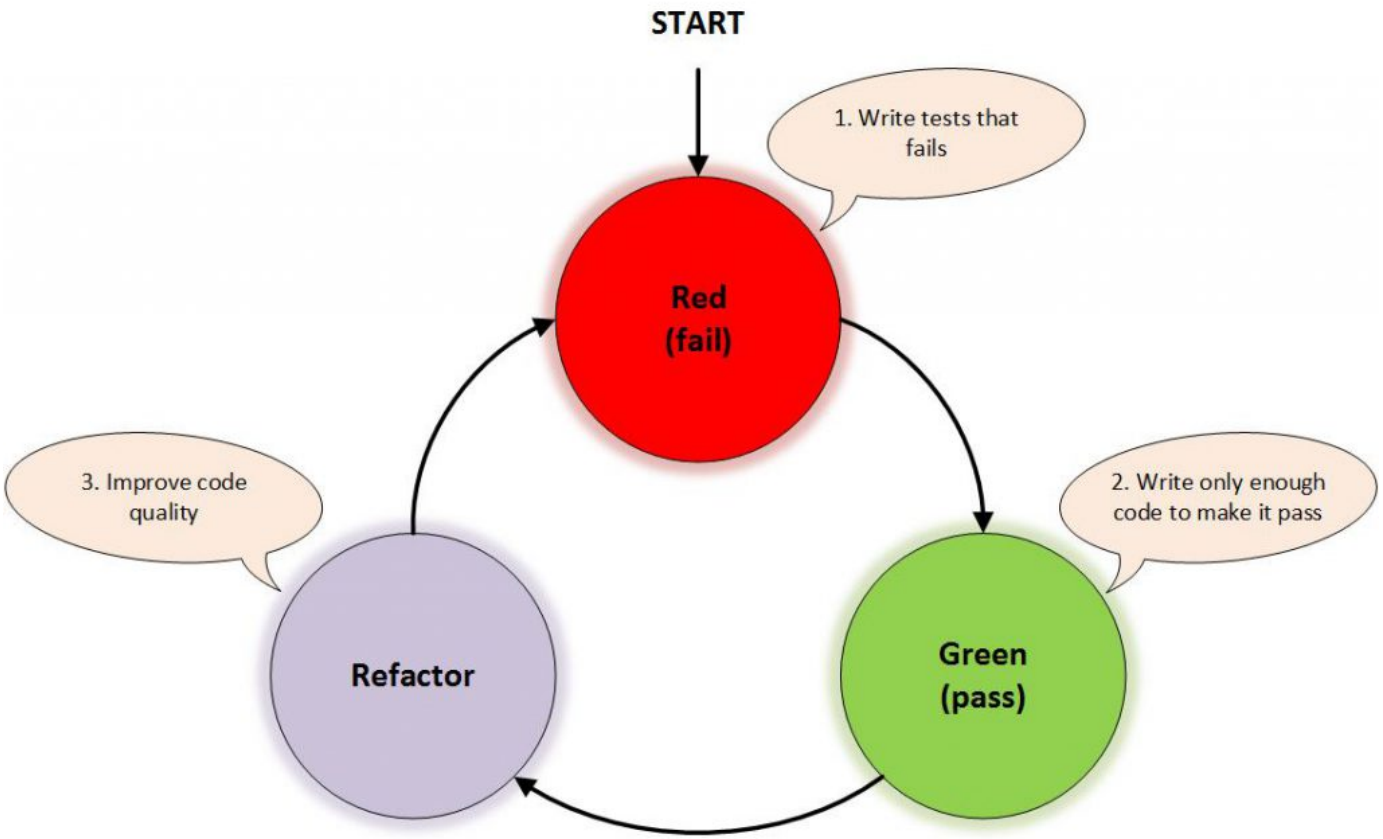

So, we're suppose to test this BEFORE we build it?

Test first or Code first?

```
test_withdraw: BOOLEAN
    local
        a: ACCOUNT
    do
        comment("test withdraw: create account & withdraw")
        create a.make (1200)
        check a.balance = 0 and a.credit = 1200 end
        a.withdraw (700)
        Result := a.balance = -700 and a.credit = 1200
    end
```

TDD
Plan & Build vs. Hack & Test

- Tests are written in advance of the code
  - Say what you want before you code
  - "How would I know that I got it right"?
- Design & Code a little at a time as you go along


START
1. Write tests that fails
Red (fail)
2. Write only enough code to make it pass
Green (pass)
Refactor
3. Improve code quality

TDD

- Think about what you want to do. Write a small test.
- Write just enough code so that your project compiles but fails the test. Think about the desired API.
- Run and watch the test fail. ("Red Bar").
- Write just enough code to pass the test and pass all your previous tests. ("Green Bar").
- Red Bar ➔ you did something wrong,
  Fix it now (it's got to be something you just wrote).
- Refactor – remove duplicate or inexpressive code, remove duplication & coupling, increase cohesion.
- Repeat

TDD

- Work with Confidence (your code constantly tested)
- Work on a series of achievable steps instead of tackling a big problem all at once
- Ensure that software design & code meets the actual requirements
  - Constant refactoring of the "design" to keep it clean
  - Fred Brook's underlying "conceptual construct"
- Leave behind a suite of tests to help preserve the integrity of the code. As a side effects you get
  - Good test coverage
  - Quality code

Building Maintainable Software: Ten Guidelines for Future-Proof Code", by Joost Visser

- Write short units of code: limit the length of methods and constructors
- Write simple units of code: limit the number of branch points per method
- Write code once, rather than risk copying buggy code
- Keep unit interfaces small by extracting parameters into objects
- Separate concerns to avoid building large classes
- Couple architecture components loosely
- Balance the number and size of top-level components in your code
- Keep your codebase as small as possible
- Automate tests for your codebase
- Write clean code, avoiding "code smells" that indicate deeper problems

What Class?

```
class ACCOUNT create
   make
feature
   balance: INTEGER
   credit: INTEGER

   withdraw(a: INTEGER)
      require
         a >= 0
      do
         balance := balance - a
      ensure
         balance = old balance - a
         credit = old credit
      end
invariant
   balance + credit >= 0
   credit >= 0
end
```

Contracts constrain the state of the attributes

Design by Contract

Will the test succeed?

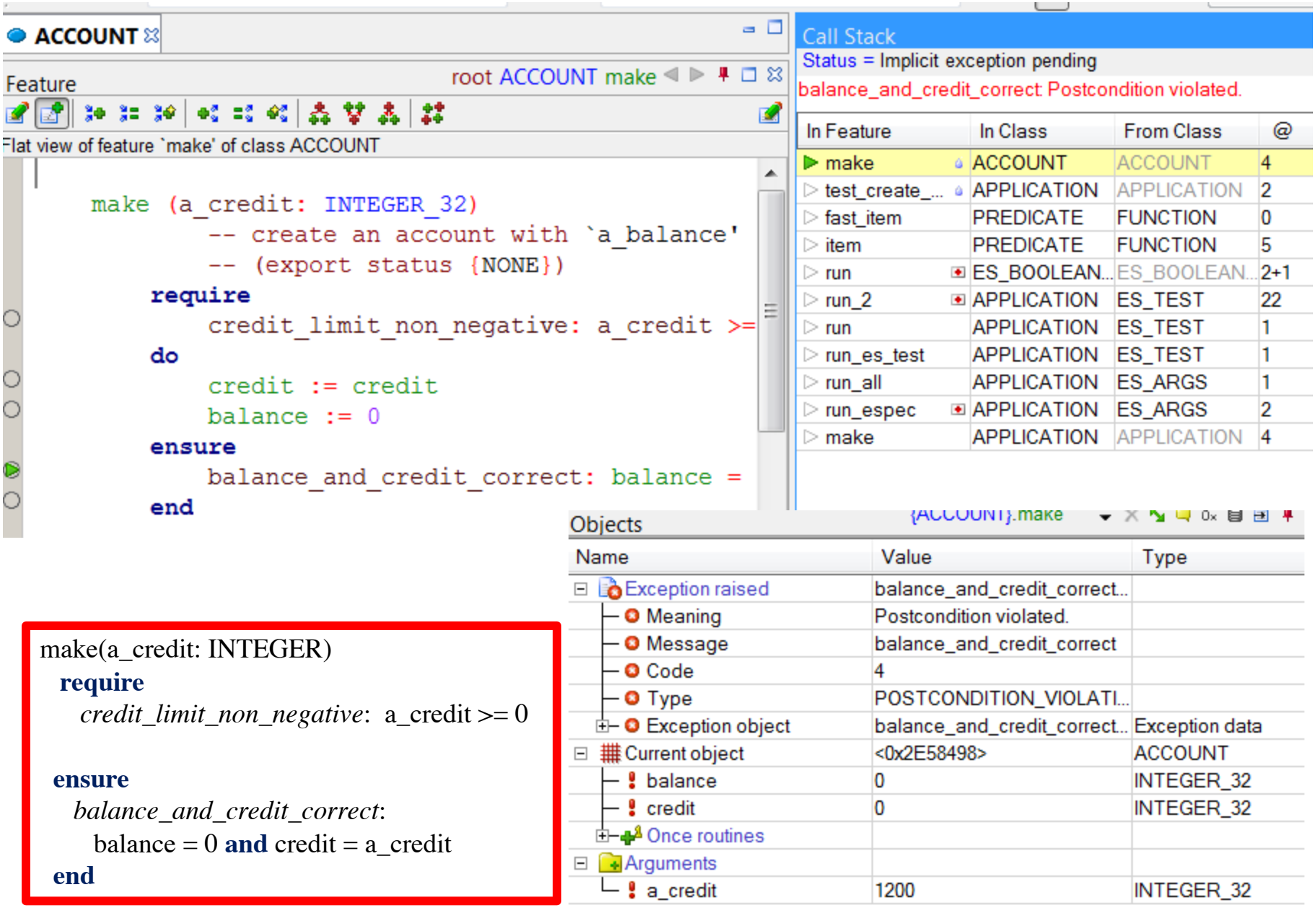
Test Run:01/05/2011 2:39:11.884 PM

Note: * indicates a violation test case

FAILED (1 failed & 0 passed out of 1)

Case Type	Passed	Total
Violation	0	0
Boolean	0	1
All Cases	0	1

State	Contract Violation	Test Name
Test1	APPLICATION	
FAILED	Postcondition violated.	t1: test_withdraw2



make(a_credit: INTEGER)
require
credit_limit_non_negative: a_credit >= 0

ensure
balance_and_credit_correct:
balance = 0 **and** credit = a_credit
end

New Test

```
test_withdraw2: BOOLEAN
    local
        a: ACCOUNT
    do
        comment("withdraw more than allowed credit")
        create a.make (1200)
        a.withdraw (1300)
        Result := a.balance = -1300
    end
```

Test Run:01/05/2011 2:45:32.904 PM

Note: * indicates a violation test case

FAILED (1 failed & 2 passed out of 3)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	3
All Cases	2	3
State	**Contract Violation**	**Test Name**
Test1	APPLICATION	
PASSED	NONE	t1: test_create_account
PASSED	NONE	t1: test_withdraw
FAILED	Class invariant violated.	t1: test_withdraw2

```
withdraw(a: INTEGER)
    require
        not_too_small: a >= 0
        not_too_big: a <= (balance + credit)
    do
        balance := balance - amount
    ensure
        balance = old balance - a
        credit = old credit
    end
```

Strengthen precondition
To preserve invariant

- Any client calling **withdraw** must first check the **precondition**
- Only if the precondition holds will the supplier guarantee the **postcondition**

Unit Tests

```
class APPLICATION inherit
  ES_TEST
create
   make
feature
   make
         -- create tests
      do
         add_boolean_case (agent test_withdraw)
         ...
         show_browser
         run_espec
      end

feature -- tests
         test_withdraw: BOOLEAN ...
end
```

Bank User Requirements

- R1: Account balances always exceed the credit limit
- R2: Clients can deposit and withdraw dollars
- **R3: Tellers (but not clients) can date a dollar deposit or withdrawal differently than today (e.g. as "tomorrow")**
- **R4: Tellers (but not clients) can access withdrawals on a given date**
- R5: Maximum total withdrawal per day is $5000
- R6: Clients can access the total amount of dollars deposited and dollars withdrawn from their account

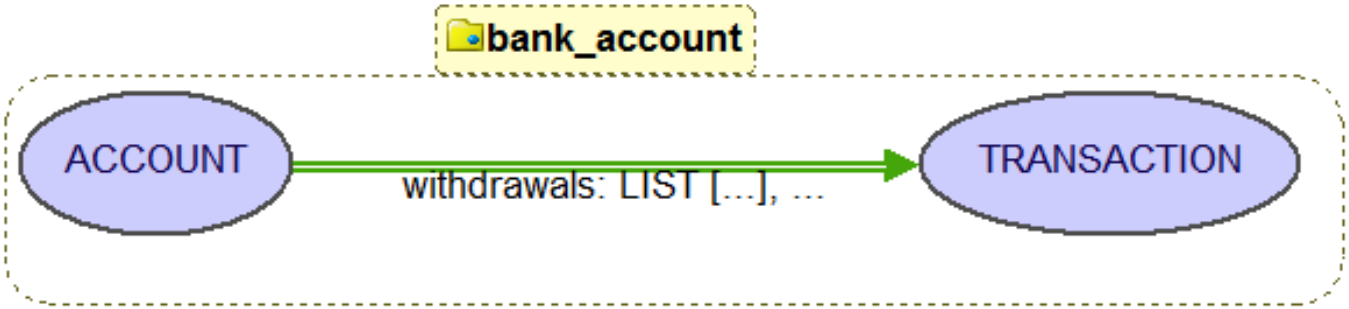
Requirements R3 and R4?

- **R3**: Tellers (but not clients) can date a dollar deposit or withdrawal differently than today (e.g. as "tomorrow")
- **R4**: Tellers (but not clients) can access withdrawals on a given date

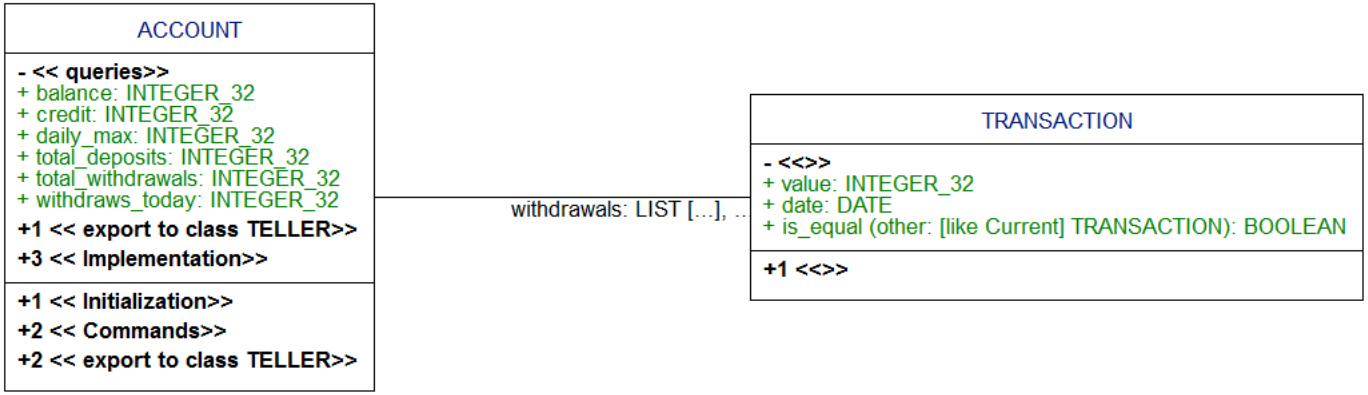
What classes or features?

- A new class →TRANSACTION
  - to store [value, date] pairs
  - Keep a list of transactions so that we can access the withdrawals on a given date

BON Diagram



UML Diagram



Generic parameters (short)

Java List of Object

```
List v = new ArrayList();
v.add("test");
Integer i =
(Integer)v.get(0);
// cast/coerce to integer
//Run time error


public interface List<G>
{ void add(G x); ...}

List<String> v =
   new ArrayList<String>();
v.add("test");
Integer i = v.get(0);
// (type error)
//Compile time error
```

Eiffel

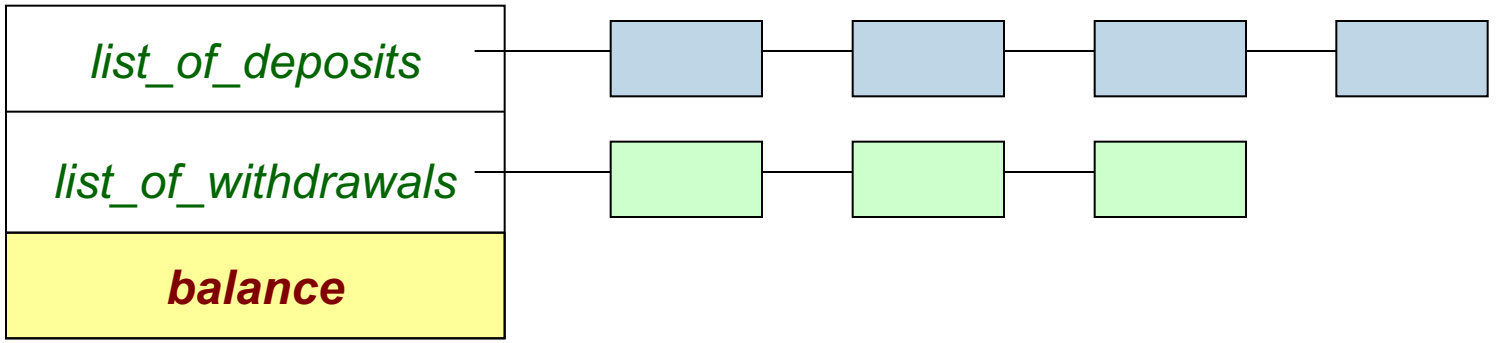
```
class LIST [G] …

   extend (x: G)
      ensure
         is_inserted (x)
      end

end



v: LIST[ STRING]
i: INTEGER

create {ARRAY_LIST[STRING]}v.make
v.extend("test")
i := v[1]
-- compile time error
```

Uniform Access Principle

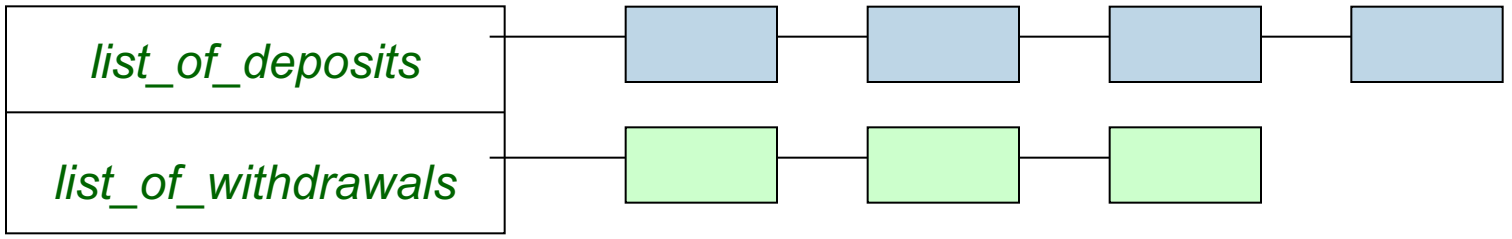
balance?

Possible instances of ACCOUNT

(A1)



(A2)



balance = total deposits – total withdrawals

Uniform Access

```
class ACCOUNT feature
 deposits: LIST[TRANSACTION]
 withdraws: LIST[TRANSACTION]

 balance: INTEGER- attribute




end
```

```
class ACCOUNT feature
 deposits: LIST[TRANSACTION]
 withdraws: LIST[TRANSACTION]

 balance: INTEGER- routine
       do
          Result := deposits.total
                       -withdrawals.total
       end
end
```

The Principle of Uniform Access

- Facilities managed by a module must be accessible to clients in the same way whether implemented by computation or storage.

- Implementation detail should not be seen at the user interface

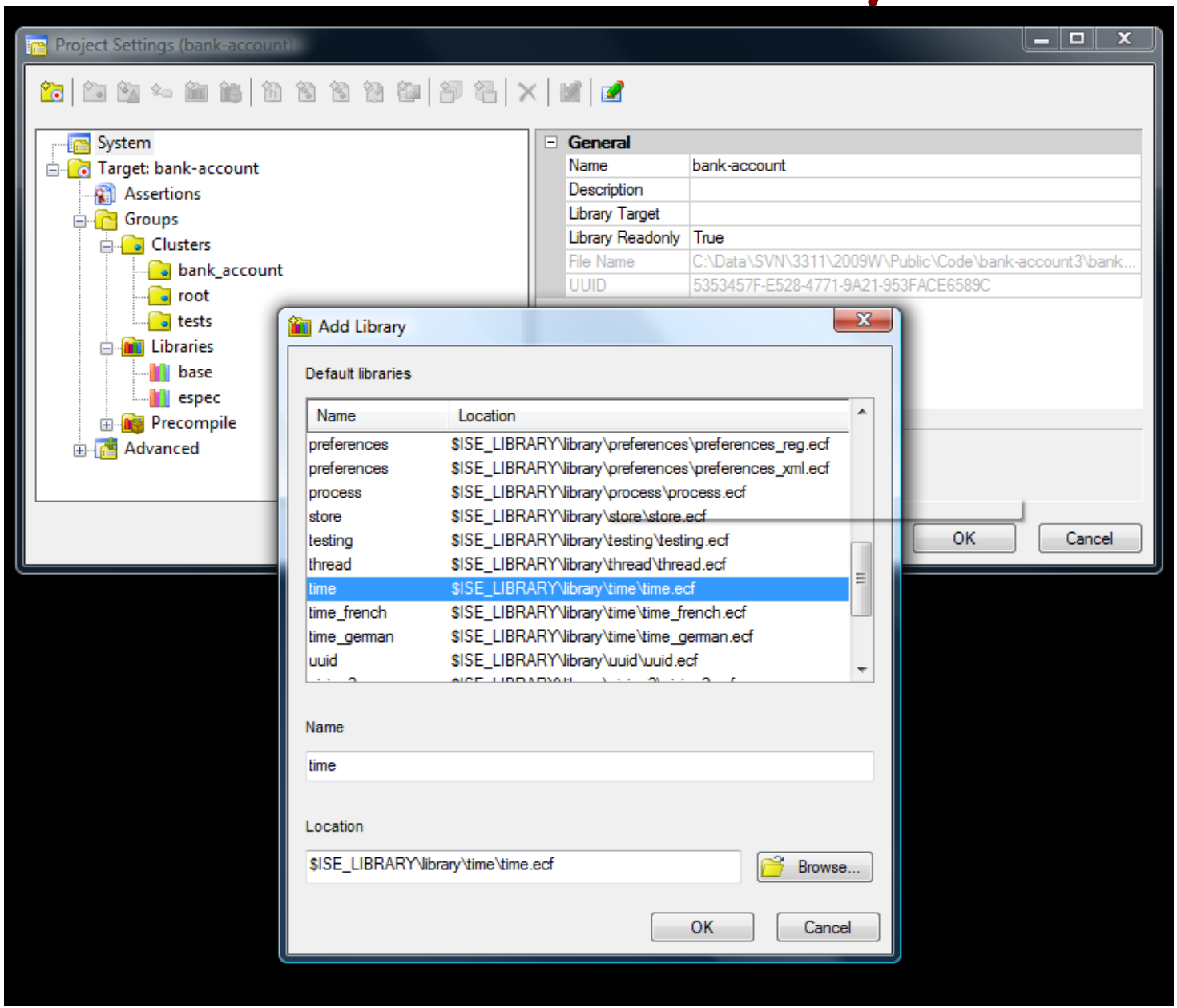
Modules needed for R3 and R4

- R3: New class TRANSACTION to store [value, date] pair
- R4: withdraws_on(d: DATE): ARRAY[TRANSACTION]
  - for the teller to query the withdrawals
- Implementation:
  LIST[TRANSACTION] to store the deposits and withdrawals

TRANSACTION

```
class TRANSACTION create
        make
feature {NONE} -- private
        make (v: INTEGER; d: DATE)
                        -- create transaction with value `v' and date `d'
                require
                        d /= Void -- not needed for void safety
                ensure
                        value_set: value = v and date = d
                end
feature -- public
        value: INTEGER

        date: DATE

invariant
        value >= 0 and date /= Void
end
```

Add time library



ECF File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<system xmlns...>
<target name="bank-account">
        <root class="APPLICATION" feature="make"/>
        <assertions precondition="true" postcondition="true"
                check="true" invariant="true" loop="true"/>
        </option>
        <precompile name="base_pre" location="$ISE_PRECOMP\base-safe.ecf"/>
        <library name="base"
                location="$ISE_EIFFEL\library\base\base-safe.ecf"/>
        <library name="espec" location="$ESPEC\library\espec-safe.ecf"/>
        <library name="time"
                location="$ISE_LIBRARY\library\time\time-safe.ecf"/>
        <cluster name="bank_account" location=".\bank-account\"/>
        <cluster name="root" location=".\root\"/>
        <cluster name="tests" location=".\tests\"/>
</target>
</system>
```

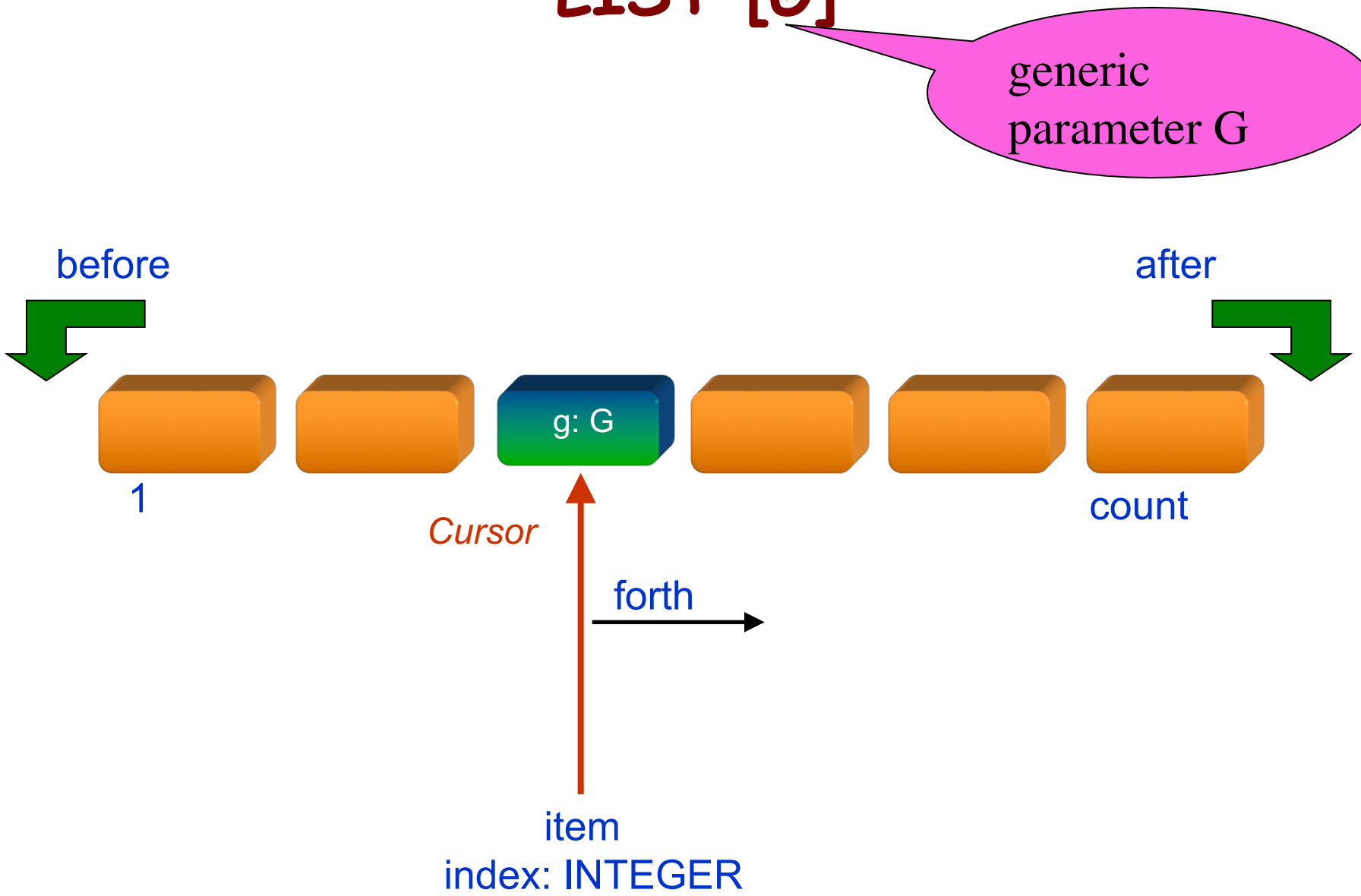
TRANSACTION (with DATE)

```
class TRANSACTION create
        make
feature {NONE}
        make (v: INTEGER; d: DATE)
                        -- create transaction with value `v' and date `d'
                require
                        d /= Void
                do
                        -- implementation goes here
                ensure
                        value_set: value = v and date = d
                end
feature
        value: INTEGER

        date: DATE

invariant
        valid_data : value >= 0 and date /= Void
end
```

Change implementation of balance

```
feature
    balance: INTEGER
        do
            -- calculate using list of deposits and withdrawals
        end
    ...

feature {NONE} -- Implementation

    deposits:     LIST [TRANSACTION]

    withdrawals: LIST[TRANSACTION]
```

LIST [G]


generic
parameter G
before
after
g: G
1
count
Cursor
forth
item
index: INTEGER

```
total_deposits: INTEGER
        do
            from deposits.start
            until
                deposits.after
            loop
                Result := Result + deposits.item.value
                deposits.forth
            end
        ensure
            comment("see invariant for postcondition")
        end
```

```
feature {NONE} -- Implementation

    deposits: LIST [TRANSACTION]

    withdrawals: LIST[TRANSACTION]
```

```
total_deposits: INTEGER
    do
        across deposits as a_transaction loop
            Result := Result + a_transaction.item.value
        end
    end
```

Inherits from ITERABLE[G]
See EiffeL101

```
feature {NONE} -- Implementation

    deposits: LIST [TRANSACTION]

    withdrawals: LIST[TRANSACTION]
```

```
deposit (a: INTEGER)
        -- deposit amount `a' into the account
    require
        not_too_small: a > 0
    local
        t: TRANSACTION; date: D
    do
                create date.make_now
                create t.make (a,date)
                deposits.extend (t)
    ensure
        balance = old balance + a and credit = old credit
        imp: deposits.count = old deposits.count + 1
    end
feature {NONE} - Implementation
        deposits: LIST [TRANSACTION]
        withdrawals: LIST[TRANSACTION]
```

original implementation:
balance := balance + a
- implementation varies
- contracts remain the same

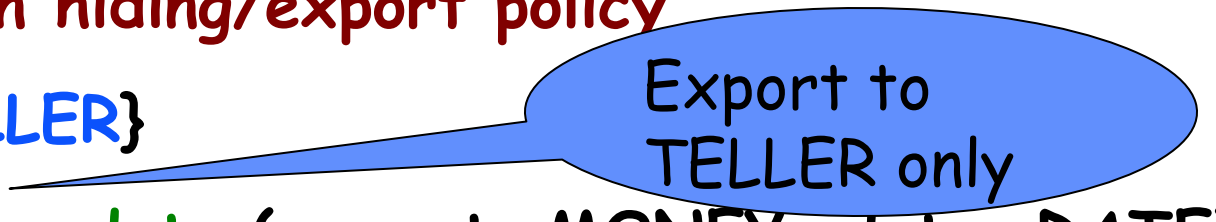
Information Hiding
- Hide private implementation from the client

New features

- withdraw_on_date ($400, tomorrow)
- deposit_on_date ($400, tomorrow)
- withdraws_today: MONEY

- Information hiding/export policy

```
feature {TELLER}
    withdraw_on_date (amount: MONEY; date: DATE)
    deposit_on_date (amount: MONEY; date: DATE)
```



New features

- withdraw_on_date (amount: MONEY; date: DATE)
  - withdraw_on_date ($400, tomorrow)
- withdraws_today: MONEY
- deposit_on_date (amount: MONEY; date: DATE)
  - deposit_on_date ($400, tomorrow)

- Write a Specification Test for ACCOUNT and TRANSACTION to describe how these features interact with each other

```
test_transaction_value_and_date: BOOLEAN
    local
        a: ACCOUNT ; today,tomorrow: DATE
        w1,w2,w3: TRANSACTION
        today_withdraws: ARRAY[TRANSACTION]
    do
        create today.make_now
        create tomorrow.make_now; tomorrow.day_forth
        create a.make (0)
        a.deposit (5500); a.withdraw (1000); a.withdraw (4000)
        a.withdraw_on_date (400, tomorrow)
        Result := a.balance = 100 and a.withdraws_today = 5000
        check Result end
        today_withdraws := a.withdraws_on (today)
        Result := today_withdraws.count = 2;
        check Result end
        create w1.make (1000, today)
        create w2.make (4000, today)
        create w3.make (400, tomorrow)
        Result := today_withdraws.has (w1) and today_withdraws.has (w2)
                                           and not today_withdraws.has (w3)
    end
```

Test is a Partial Specification

Specification of 3 new features

Chart View of ACCOUNT

class

ACCOUNT

Queries

balance: INTEGER
credit: INTEGER
Daily_max: INTEGER
total_deposits: INTEGER
total_withdrawals: INTEGER
withdraws_today: INTEGER

Commands

deposit (a: INTEGER)
withdraw (a: INTEGER)

Constraints

positive balance: balance + credit >= 0
balance_consistent: balance = total_deposits - total_withdrawals

Exported to TELLER

deposit_on_date (a: INTEGER; a_date: DATE)
withdraw_on_date (a: INTEGER; date: DATE)
withdraws_on (d: DATE): ARRAY [TRANSACTION]

```
class
        ACCOUNT
Queries
        balance: INTEGER
        credit: INTEGER
        Daily_max: INTEGER
        total_deposits: INTEGER
        total_withdrawals: INTEGER
        withdraws_today: INTEGER

Commands
        deposit (a: INTEGER)
        withdraw (a: INTEGER)

Constraints
        positive balance: balance + credit >= 0
        balance_consistent: balance = total_deposits - total_withdrawals
```

```
Exported to TELLER

        deposit_on_date (a: INTEGER; a_date: DATE)
        withdraw_on_date (a: INTEGER; date: DATE)
        withdraws_on (d: DATE): ARRAY [TRANSACTION]

Private
        deposits:    LIST[TRANSACTION]
        withdrawals: LIST[TRANSACTION]
```

```
test_transaction_value_and_date: BOOLEAN
    local
        a: ACCOUNT ; today,tomorrow: DATE
        w1,w2,w3: TRANSACTION
        today_withdraws: ARRAY[TRANSACTION]
    do
        create today.make_now
        create tomorrow.make_now; tomorrow.day_forth
        create a.make (0)
        a.deposit (5500); a.withdraw (1000); a.withdraw (4000)
        a.withdraw_on_date (400, tomorrow)
        Result := a.balance = 100 and a.withdraws_today = 5000
        check Result end
        today_withdraws := a.withdraws_on (today)
        Result := today_withdraws.count = 2;
        check Result end
        create w1.make (1000, today)
        create w2.make (4000, today)
        create w3.make (400, tomorrow)
        Result := today_withdraws.has (w1) and today_withdraws.has (w2)
                                           and not today_withdraws.has (w3)
  end
```

```
withdraw_on_date (amount: INTEGER; date: DATE)
        -- withdraw `amount` dated `date`
    require
        amount > 0 and (balance + credit – amount) ≥  0
        date /= Void
    local
        t: TRANSACTION
    do
        create t.make (amount,date)
        withdrawals.extend (t)
    ensure
        (balance=old balance–amount) and credit=old credit
        withdrawals.count = old withdrawals.count + 1
    end
 feature {NONE} – Implementation
        deposits:     LIST [TRANSACTION]
        withdrawals: LIST[TRANSACTION]
```

FAILED (1 failed & 7 passed out of 8)

Case Type	Passed	Total
Violation	2	2
Boolean	5	6
All Cases	7	8

State	Contract Violation	Test Name
Test1	TEST_ACCOUNT	
PASSED	NONE	t0: test_create_account
PASSED	NONE	t2: test_deposit
PASSED	NONE	t1: test_withdraw
PASSED	NONE	t4: test_deposit_and_withdraw
PASSED	NONE	t5: test_transaction_value_and_date
FAILED	Feature call on void target.	t6: test_transaction_value_and_date2
PASSED	NONE	***t100: test_withdraw_violation
PASSED	NONE	***t101: test_deposit_amount violation

```
test_transaction_value_and_date2: BOOLEAN
        -- not a query; no return type
    local
        a: ACCOUNT
        today, tomorrow: DATE
        today_withdraws: ARRAY [TRANSACTION]
        w1, w2, w3: TRANSACTION
    do
        comment ("t6: test_transaction_value_and_date2")
        create today.make_now
        create tomorrow.make_now
        tomorrow.day_forth
        create a.make (0)
        a.deposit (5500)
        check
                a.balance = 5500
        end
        a.withdraw (1000)
        a.withdraw_on_date (400, tomorrow)
        a.withdraw (4000)
        Result := a.balance = 100
        check
                Result
        end
```

$1000 today
$400 tomorrow
$4000 today

Feature call on Void Target

a.withdraw must satisfy an invariant in ACCOUNT

```
invariant
   0 <= withdraws  today
   withdraws_today <= daily_max
```

withdraws_today calls withdraws_on(today) which as we shall see is implemented incorrectly

```
withdraws_on(d: DATE): ARRAY[TRANSACTION]

    require d /= Void
    do




    ensure
        -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
    end
```

class ACCOUNT uses ARRAY

```
class ARRAY[G] feature
  lower, upper, count: INTEGER

  put (v: G; i: INTEGER)
        -- Replace `i'-th entry, if in index interval, by `v'.
      require
        valid_key: valid_index (i)

  force (v: G; i: INTEGER)
        -- Assign item `v' to `i'-th entry.
        -- Resize the array if `i' falls out of
        -- currently defined bounds; preserve existing items.

  ...
end
```

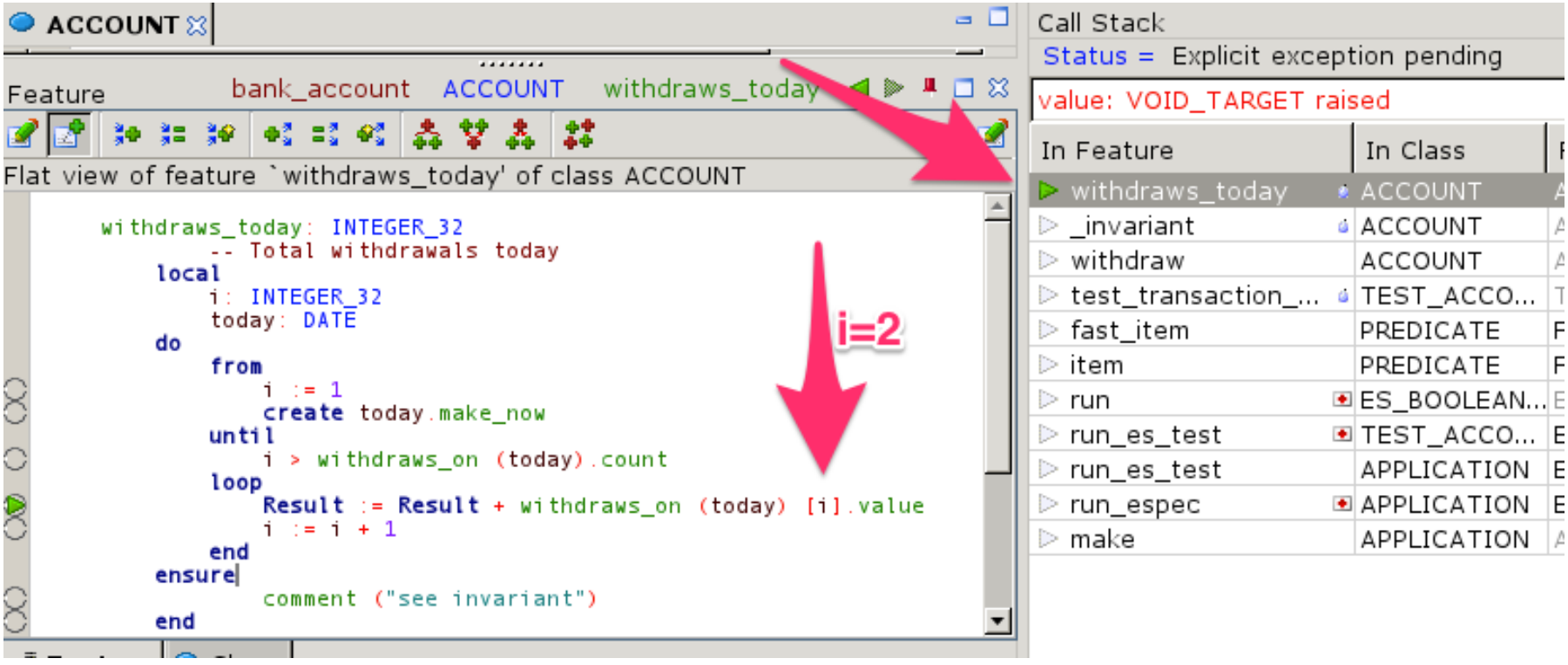
```
withdraws_on(d: DATE): ARRAY[TRANSACTION]
    require d /= Void
    local t: TRANSACTION; i: INTEGER
    do
        create Result.make_empty
        from
            withdrawals.start; i := 1
        until
            withdrawals.after
        loop
            if  withdrawals.item.date  ~  d then
                t := withdrawals.item
                Result.force (t, i)
            end
            i := i + 1
            withdrawals.forth
        end
    ensure
        -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
    end
```

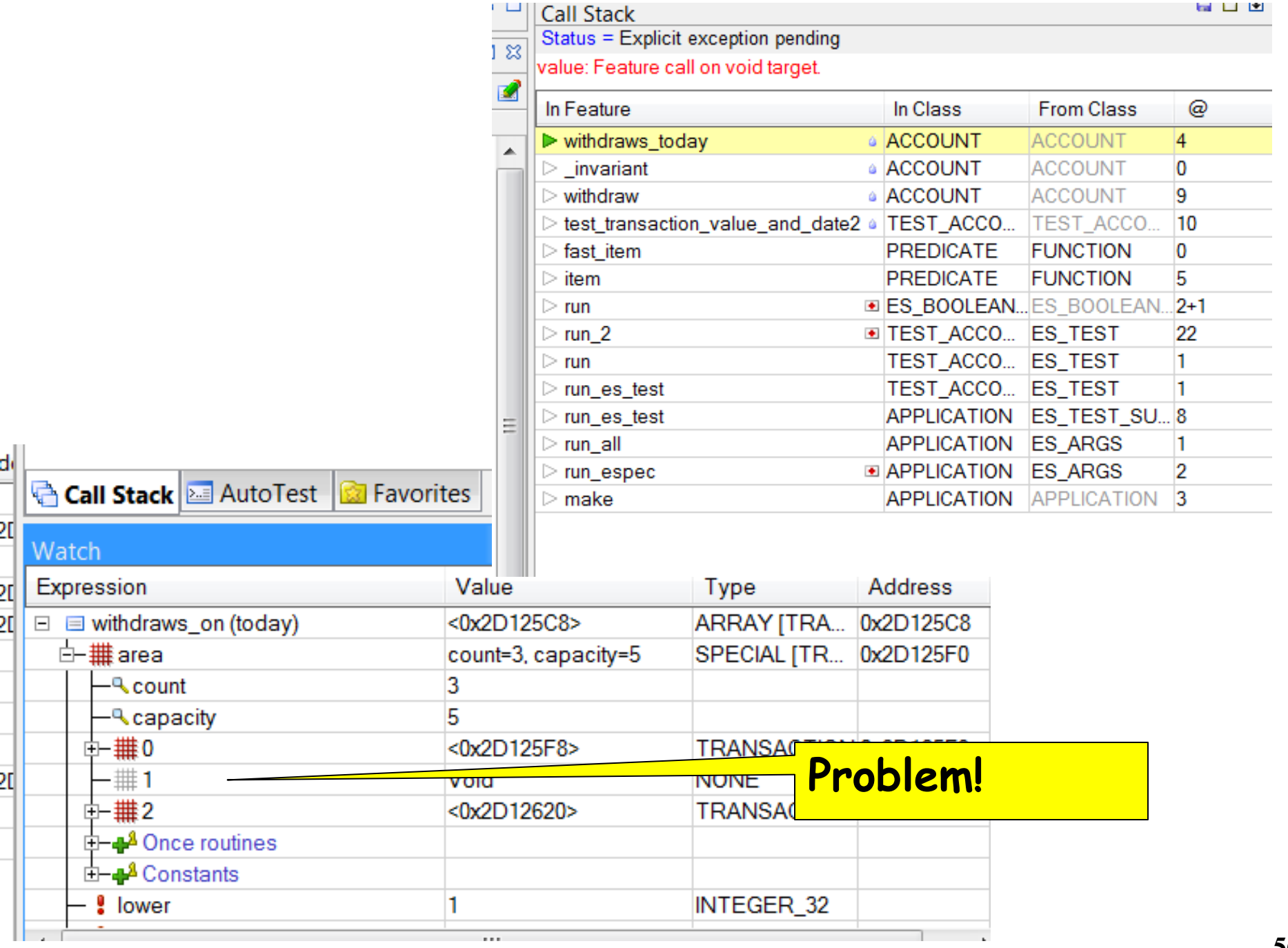
Can you spot the problem?

Problem!

Void safety required

invariant:
0 <= withdraws_today
and withdraws_today <= daily_max




Call Stack
Status = Explicit exception pending
value: Feature call on void target.
In Feature
In Class
From Class
@
withdraws_today ACCOUNT ACCOUNT 4
_invariant ACCOUNT ACCOUNT 0
withdraw ACCOUNT ACCOUNT 9
test_transaction_value_and_date2 TEST_ACCO... TEST_ACCO... 10
fast_item PREDICATE FUNCTION 0
item PREDICATE FUNCTION 5
run ES_BOOLEAN... ES_BOOLEAN... 2+1
run_2 TEST_ACCO... ES_TEST 22
run TEST_ACCO... ES_TEST 1
run_es_test TEST_ACCO... ES_TEST 1
run_es_test APPLICATION ES_TEST_SU... 8
run_all APPLICATION ES_ARGS 1
run_espec APPLICATION ES_ARGS 2
make APPLICATION APPLICATION 3
Call Stack
AutoTest
Favorites
Watch
Expression
Value
Type
Address
withdraws_on (today) <0x2D125C8> ARRAY [TRA... 0x2D125C8
area count=3, capacity=5 SPECIAL [TR... 0x2D125F0
count 3
capacity 5
0 <0x2D125F8>
1
2 <0x2D12620>
Once routines
Constants
lower 1 INTEGER_32
Problem!

withdraws_on(today)[2] = Void

```
withdraws_today: INTEGER
        -- Total withdrawals today
    local
        i: INTEGER
        today: DATE
    do
        from
            i := 1
            create today.make_now
        until
            i > withdraws_on (today).count
        loop
            Result := Result + withdraws_on(today)[i].value
            i := i + 1
        end
    ensure
        comment("see invariant")
    end
```

```
withdraws_on(d: DATE): ARRAY[TRANSACTION]
    local
        t: TRANSACTION; i: INTEGER
    do
        create Result.make_empty
        from
            withdrawals.start; i := 1
        until
            withdrawals.after
        loop
            if  withdrawals.item.date  ~  d then
                t := withdrawals.item
                Result.force (t, i)
                i := i + 1
            end
            withdrawals.forth
        end
    ensure
        -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
    end
```

Problem fixed!

```
withdraws_on(d: DATE): ARRAY[TRANSACTION]
      -- return all withdrawals on date `d`
    local
       t: TRANSACTION; i: INTEGER
    do
       create Result.make_empty -- Result.compare_objects
       from withdrawals.start; i := 1
       until withdrawals.after
       loop
          if withdrawals.item.date ~ d then
             t := withdrawals.item
             Result.force (t, i); i := i + 1
          end
          withdrawals.forth
       end
    ensure
       -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
       across withdrawals as t all
          Result.has(t.item) = (t.item.date ~ d)
       end
    end
```

This is how you should document your own code
OOSC2 Chapter 26 A sense of Style

```
withdraws_on(d: DATE): ARRAY[TRANSACTION]
      -- return all withdrawals on date `d`
   local
      t: TRANSACTION; i: INTEGER
   do
      create Result.make_empty -- Result.compare_objects
      from withdrawals.start; i := 1
      until withdrawals.after
      loop
         if withdrawals.item.date ~ d then
            t := withdrawals.item
            Result.force (t, i); i := i + 1
         end
         withdrawals.forth
      end
   ensure
      -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
      across withdrawals as t all
         Result.has(t.item) = (t.item.date ~ d)
      end
   end
```

Test Run:03/17/2009 5:36:02.996 PM

APPLICATION

Note: * indicates a violation test case

FAILED (1 failed & 6 passed out of 7)		
Case Type	**Passed**	**Total**
Violation	2	2
Boolean	4	5
All Cases	6	7
State	**Contract Violation**	**Test Name**
Test1	TEST_ACCOUNT	
PASSED	NONE	t0: test_create_account
PASSED	NONE	t2: test_deposit
PASSED	NONE	t1: test_withdraw
PASSED	NONE	t4: test_deposit_and_withdraw
FAILED	NONE	t5: test_transaction_value_and_date
PASSED	NONE	**t100: test_withdraw_violation
PASSED	NONE	**t101: test_deposit_amount violation

Further problems ☹

- Failure must be at the last line (why?)
- Last line is:

```
today_withdraws: ARRAY[TRANSACTION]
today_withdraws := a.withdraws_on (today)
...
Result :=       today_withdraws.has(w1)
        and     today_withdraws.has(w2)
        and not today_withdraws.has(w3)
```

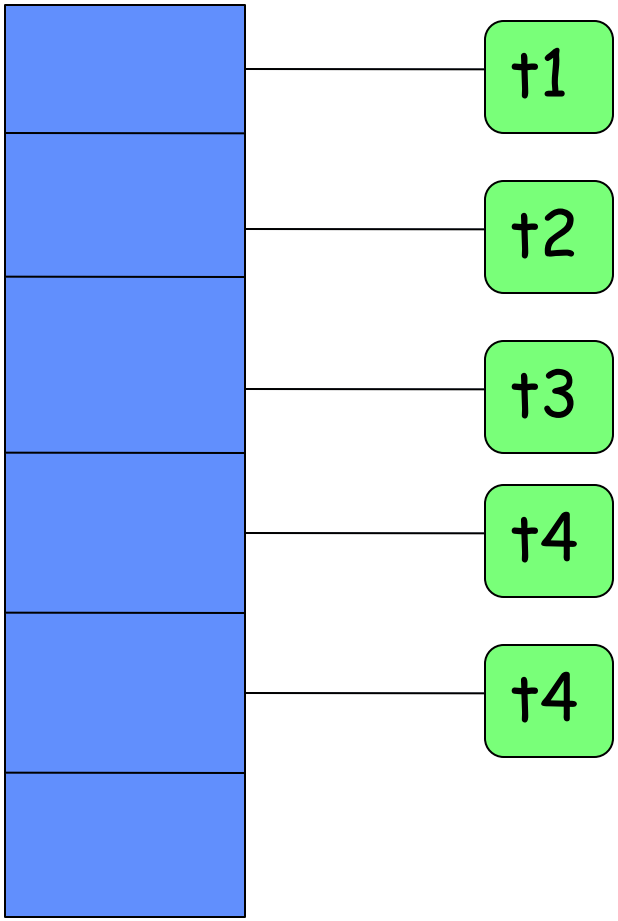
- Why does the test fail at the last line?
- How do you determine if a transaction object is in the array? withdraws_today.has (w1)
- Main problem has to do with object vs. reference comparisons in the array of transactions

Object vs. Reference Comparisons

```
class ARRAY [G] inherit ...
   redefine is_equal end
feature
   has (v: G): BOOLEAN is
         -- Does `v' appear in array?
         -- based on `object_comparison'

   object_comparison: BOOLEAN
         -- Must search operations use `equal' rather than `='
         -- Default: false, i.e. use `='.

   compare_objects
         -- Ensure that future search operations will use `equal'
         -- rather than `=' for comparing references.

   compare_references
         -- Ensure that future search operations will use `='
         -- rather than `equal' for comparing references.
```

Does the array have y?

transactions: ARRAY[TRANSACTION]




```
check y = t2 end
transactions.has(y)?
```

```
check y ~ t2 end
transactions.has(y)?
```

What does it mean for the array to have element `v'?

```
has (v: G): BOOLEAN is
        -- Does `v' appear in array?
    do
        if object_comparison and v /= Void then
            from until ... loop
                ...
                Result := item.is_equal (v)
                i := i + 1
            end
        else
                ...    Result := item (i) = v
        end
    end
```

Object comparison

Reference comparison

```
withdraws_on(d: DATE): ARRAY[TRANSACTION]
    local
        t: TRANSACTION; i: INTEGER
    do
        create Result.make_empty;
        Result.compare_objects
        from
            t.start; i := 1
        until
            t.after
        loop
            if  t.item.date ~ d then
                t := withdrawals.item
                Result.force (t, i)
                 i := i + 1
            end
            withdrawals.forth
        end
    ensure
        -- ∀t ∈ withdrawals: t ∈ Result ≡ t.date=d
    end
```

Fixing the problem

equal(t.item,d)

Unfortunately, the test still fails! Why?

Test Run:03/17/2009 5:36:02.996 PM

APPLICATION

Note: * indicates a violation test case

FAILED (1 failed & 6 passed out of 7)

Case Type	Passed	Total
Violation	2	2
Boolean	4	5
All Cases	6	7

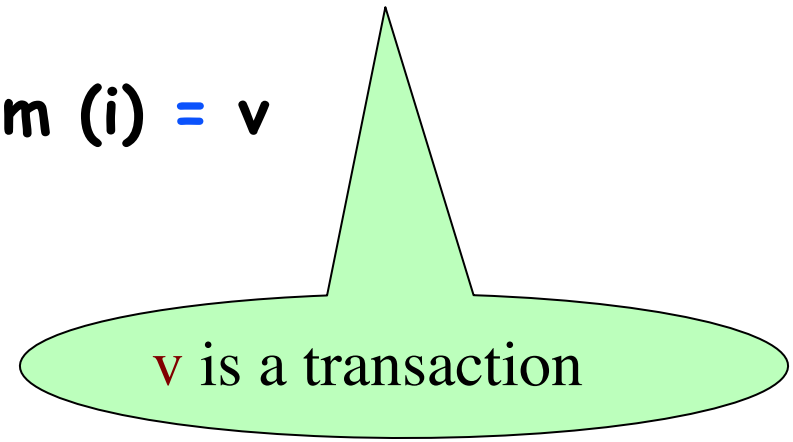
State	Contract Violation	Test Name
Test1	TEST_ACCOUNT	
PASSED	NONE	t0: test_create_account
PASSED	NONE	t2: test_deposit
PASSED	NONE	t1: test_withdraw
PASSED	NONE	t4: test_deposit_and_withdraw
FAILED	NONE	t5: test_transaction_value_and_date
PASSED	NONE	**t100: test_withdraw_violation
PASSED	NONE	**t101: test_deposit_amount violation

Specification Test – Object Comparison

```
test_transaction_value_and_date: BOOLEAN
    local
     a: ACCOUNT ; today,tomorrow: DATE
     w1,w2,w3: TRANSACTION
     today_withdraws: ARRAY[TRANSACTION]
     do
         ...
              today_withdraws := a.withdraws_on (today)

            create w1.make (1000, today)
            create w2.make (4000, today)
            create w3.make (400, tomorrow)

            Result := today_withdraws.has (w1) and
                today_withdraws.has (w2) and not
                today_withdraws.has (w3)
    end
```

How does {ARRAY}has work?

```
class ARRAY [G] ...

 has (v: G): BOOLEAN is
                    -- Does `v' appear in array?
       do
             if object_comparison and v /= Void then

                   Result := item.is_equal (v)
             else
                   Result := item (i) = v
             end
       end
```



- {ARRAY} has(t) calls {TRANSACTION} t.is_equal
- We need to define how two transactions are compared via is_equal
- We study ARRAY first

Approximate definition

- target1.is_equal(target2)
  - target1 and target2 are NOT allowed to be void
- equal(target1, target2)
  - target1 and target2 ARE allowed to be void

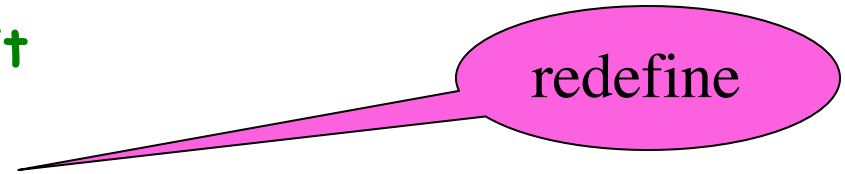
$$equal(a,b) \Leftrightarrow a \sim b$$

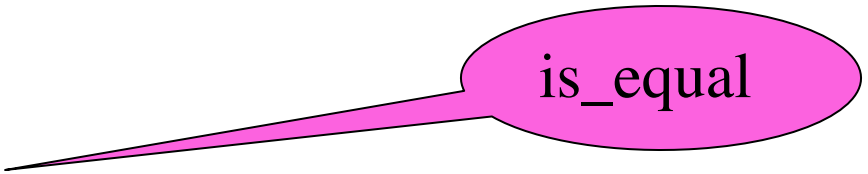
- a ~ b
  - (a = Void and then b = Void)
    or else (a /= Void and then b /= Void
      and then a.is_equal (b))

- Reference comparison
  - target1 = target2
- Object comparison
  - target1 ~ target2
- For expanded classes like INTEGER, BOOLEAN etc.
  - = and ~ are the same
  - both are object comparison, i.e. the comparison is by value
- target1 := target2
  - depends on whether the targets refer to reference or expanded objects
- All classes inherit from ANY

Redefine is_equal

```
class TRANSACTION inherit
    ANY
        redefine is_equal end
    ...
feature
    value: INTEGER
    date: DATE

    is_equal(other: like Current): BOOLEAN
        do
            Result := value = other.value and date ~ other.date
        end

invariant
    value >= 0 and date /= Void
end
```





What does it mean for 2 arrays to be equal?

```
is_equal (other: like Current): BOOLEAN
        -- Is array made of the same items as `other'?
    local
        i: INTEGER
    do
        if other = Current then
            Result := True
        elseif lower = other.lower and then upper = other.upper and then
            object_comparison = other.object_comparison
        then
            if object_comparison then
                from
                    Result := True
                    i := lower
                until
                    not Result or i > upper
                loop
                    Result := item (i) ~ other.item (i)
                    i := i + 1
                end
            else
                Result := area.same_items (other.area, 0, 0, count)
            end
        end
    end
```

Result := equal (item (i), other.item (i))

Fixes

- redefine {TRANSACTION}.is_equal
- Fixes in withdraw_on (d: DATE)
  - Result.compare_objects
  - For information hiding:
    - t := withdrawals.item.deep_twin

- See Eiffel101 Section 11

Specification Test – Object Comparison

```
test_transaction_value_and_date: BOOLEAN
    local
     a: ACCOUNT ; today,tomorrow: DATE
     w1,w2,w3: TRANSACTION
     today_withdraws: ARRAY[TRANSACTION]
     do
         ...
              today_withdraws := a.withdraws_on (today)

            create w1.make (1000, today)
            create w2.make (4000, today)
            create w3.make (400, tomorrow)

            Result := today_withdraws.has (w1) and
                today_withdraws.has (w2) and not
                today_withdraws.has (w3)
    end
```

PASSED (8 out of 8)

Case Type	Passed	Total
Violation	2	2
Boolean	6	6
All Cases	8	8

State	Contract Violation	Test Name
Test1	TEST_ACCOUNT	
PASSED	NONE	t0: test_create_account
PASSED	NONE	t2: test_deposit
PASSED	NONE	t1: test_withdraw
PASSED	NONE	t4: test_deposit_and_withdraw
PASSED	NONE	t5: test_transaction_value_and_date
PASSED	NONE	t6: test_transaction_value_and_date2
PASSED	NONE	***t100: test_withdraw_violation
PASSED	NONE	***t101: test_deposit_amount violation

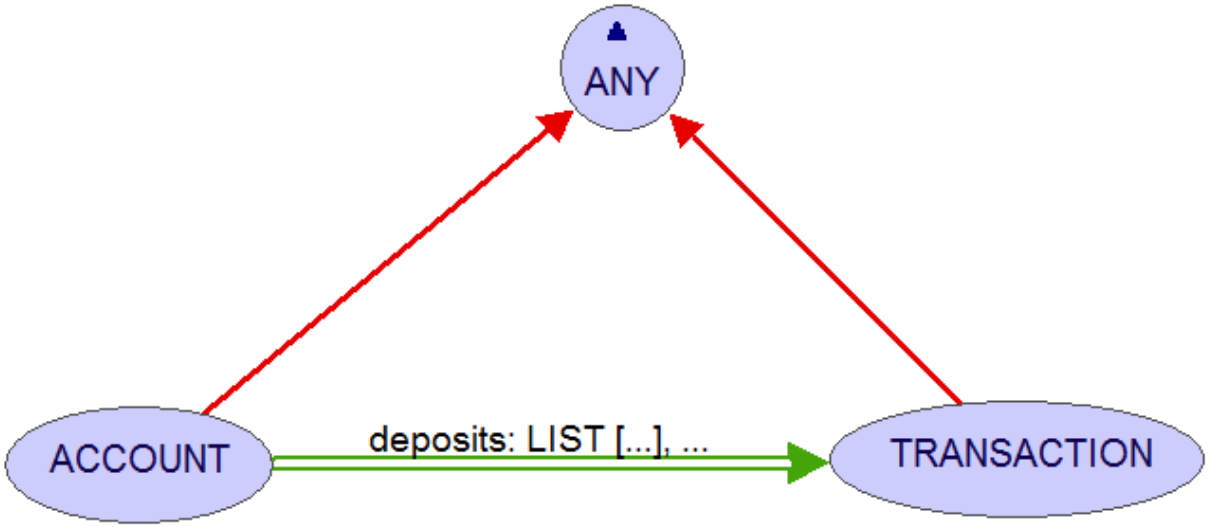
Advice

- Try out all the examples such as the bank for yourself
- Don't restrict yourself only to what is graded

- Even as languages and technologies evolve we will still be talking specifications: preconditions, postconditions, invariants.
  - Testing
  - Abstraction
  - Information Hiding

What is Design?

- What classes will satisfy Requirements?
  - ACCOUNT, TRANSACTION, TELLER ...
- What features are needed?
  - balance, deposit, withdraw ...
- How are the classes organized (via association, inheritance etc.)?



OOSC2 Chapter 22

Class Consistency principle

All the features of a class must pertain to a single, well-identified abstraction.

ADT specification of stacks

TYPES

- *STACK* [*G*]

FUNCTIONS

- *put*: *STACK* [*G*] × *G* → *STACK* [*G*]
- *remove*: *STACK* [*G*] ↛ *STACK* [*G*]
- *item*: *STACK* [*G*] ↛ *G*
- *empty*: *STACK* [*G*] → *BOOLEAN*
- *new*: *STACK* [*G*]

AXIOMS

For any *x*: *G*, *s*: *STACK* [*G*]

A1 • *item* (*put* (*s*, *x*)) = *x*

A2 • *remove* (*put* (*s*, *x*)) = *s*

A3 • *empty* (*new*)

A4 • **not** *empty* (*put* (*s*, *x*))

PRECONDITIONS

- *remove* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)
- *item* (*s*: *STACK* [*G*]) **require not** *empty* (*s*)

OOSC2 Chapter 22

Class Consistency principle

All the features of a class must pertain to a single, well-identified abstraction.

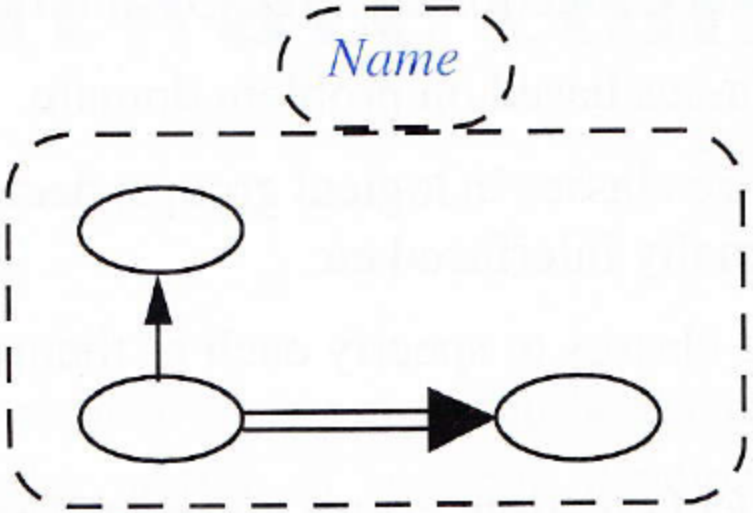
Abstract properties can be stated, informally or (preferably) formally, describing: how the results of the various queries relate to each other (this will yield the invariant); under what conditions features are applicable (preconditions); how command execution affects query results (postconditions).

OOSC2 Chapter 22

Danger signal	Why suspicious
Class with verbal name (infinitive or imperative)	• May be a simple subroutine, not a class.
Fully effective class with only one exported routine	• May be a simple subroutine, not a class.
Class described as "performing" something	• May not be a proper data abstraction.
Class with no routine	• May be an opaque piece of information, not an ADT. Or may be an ADT, the routines having just been missed.
Class introducing no or very few features (but inherits features from parents)	• May be a case of "taxomania".
Class covering several abstractions	• Should be split into several classes, one per abstraction

BON diagrams

Cluster (with some classes)

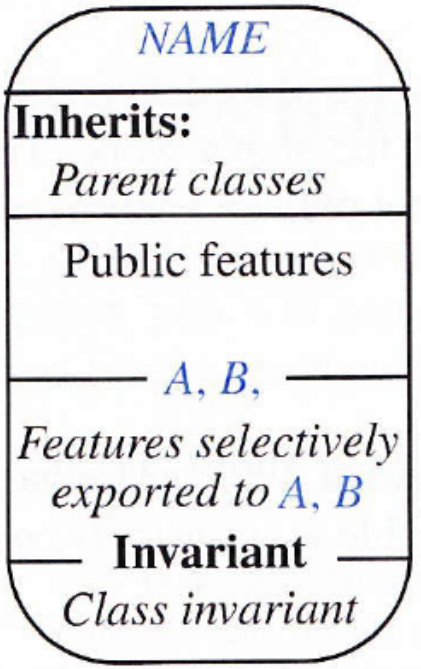


nter-class relations

Inherits from

Client — *name* →

Class: detailed interface



Features

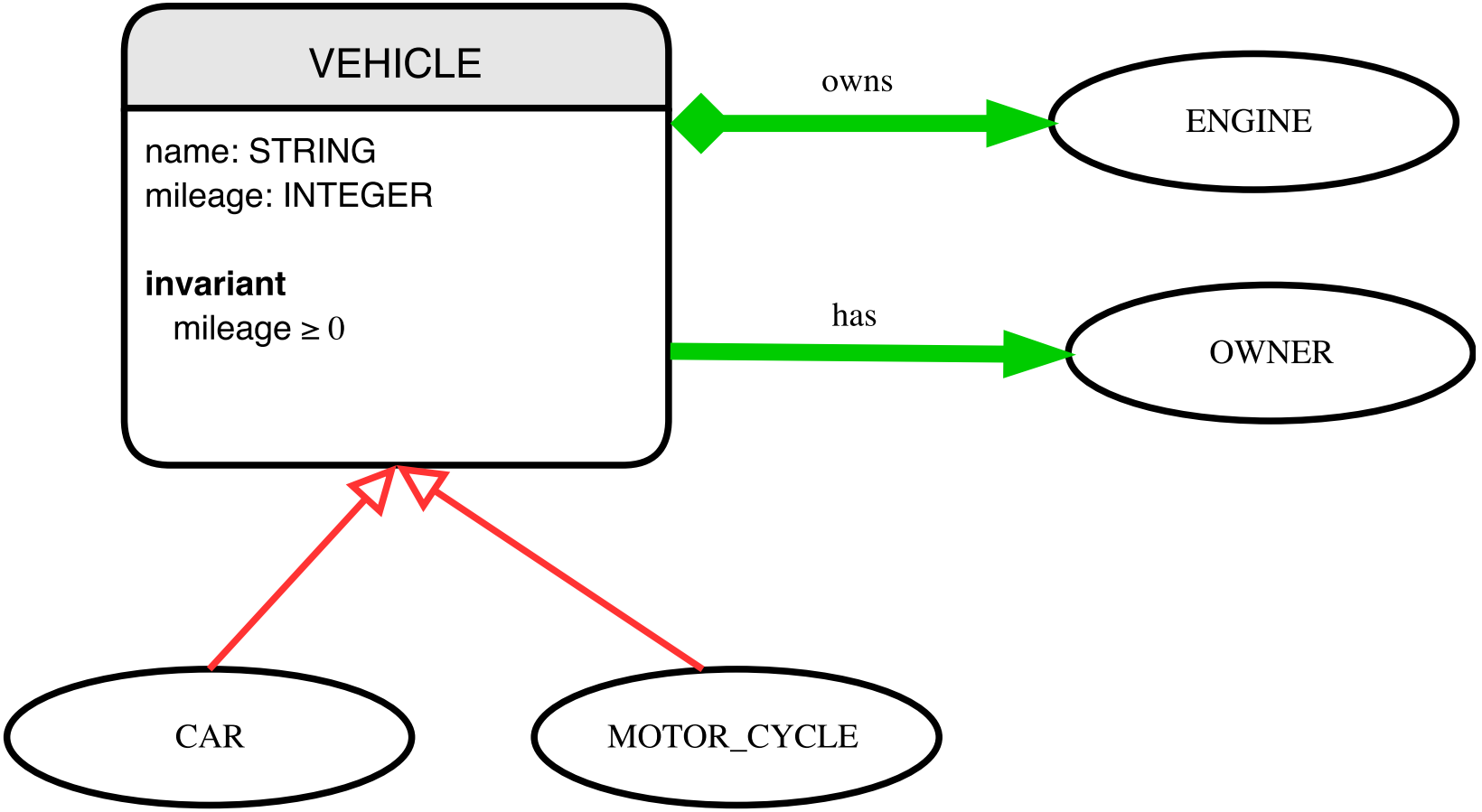
*name**, *name*$^+$, *name*$^{++}$	deferred, effective, redefined
→ *name*: *TYPE*	input argument
? !	precondition, postcondition

Assertion operators

Δ *name*	feature may change attribute *name*
@, ∅	current object, void reference
∃, ∀, |, •	symbols for predicate calculus operations
∈, ∉	membership operators

draw.io (see template)

Note clarity of drawing using PDFs



ABSTRACT_ACCOUNT

balance, credit: VALUE

withdraw (a: VALUE)
  require a > 0 ∧ a ≤ (balance + credit)
  ensure balance = **old** balance - a

invariant
balance + credit ≥ 0

Mathematical notation

ΣΠαβΨψλπ∴⊕⊗□→↔↑↓≤≥÷~≈≠
∈∉∧∨¬∩∪∀∃∅⇒⇔⊂⊄ ⟨⟩ ↖ ↦ ↾
≔≝≜≡ℤℕℝ⊨⊢•×✓⟦⟧ ⟨ ⟩ ⟪ ⟫

Better Design?

ACCOUNT

balance: VALUE++

withdraw (a: VALUE)++

feature -- { TELLER }
withdraw_on_date (amount: VALUE; date: DATE)

withdraws_on(d: DATE): ARRAY[TRANSACTION]

feature {NONE} -- implementation
deposits: LIST [TRANSACTION]
withdrawals: LIST[TRANSACTION]

withdraws_on(d:DATE): ARRAY[..]

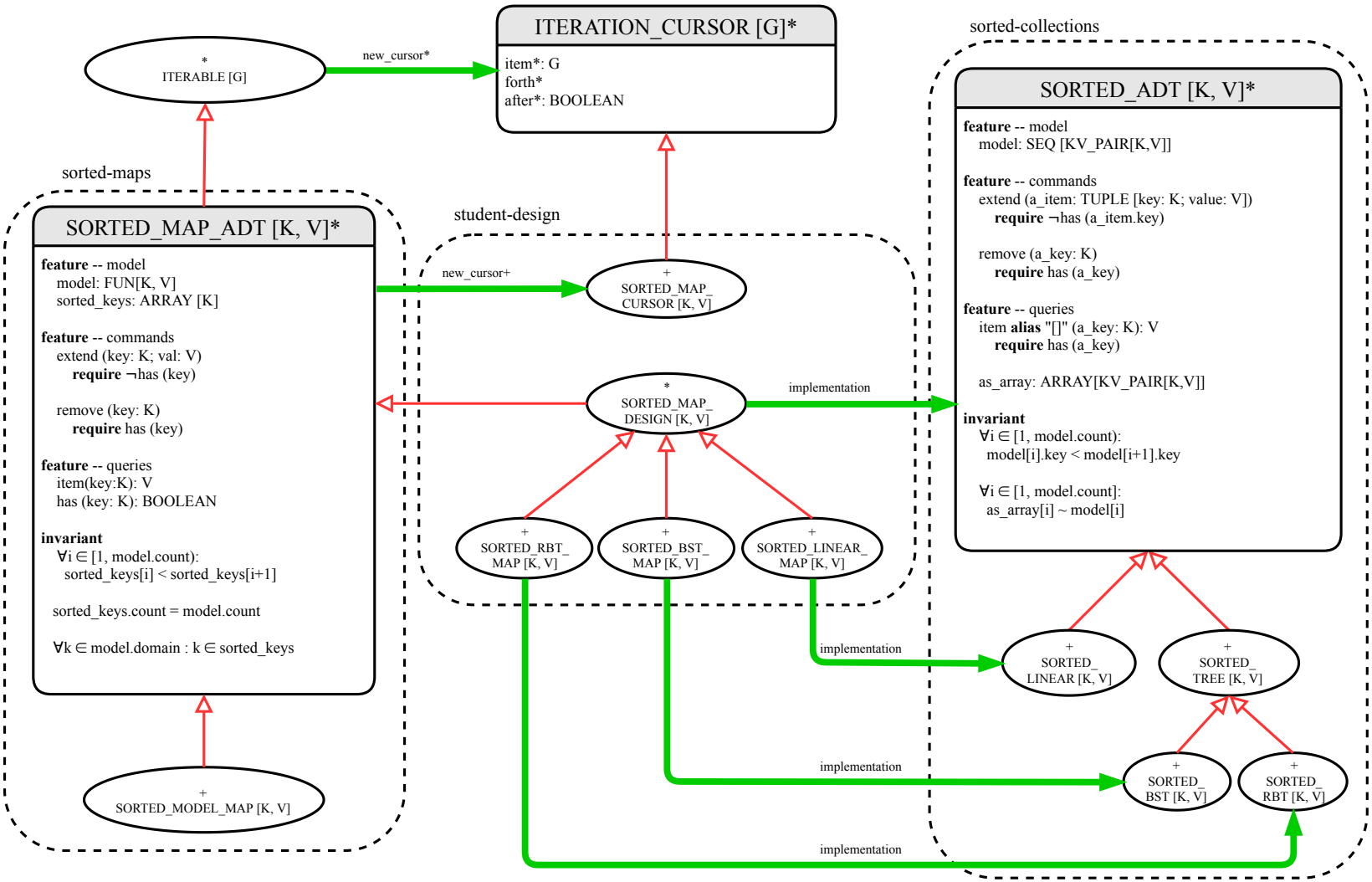
TRANSACTION

deposits: LIST[..]

What is Architecture?

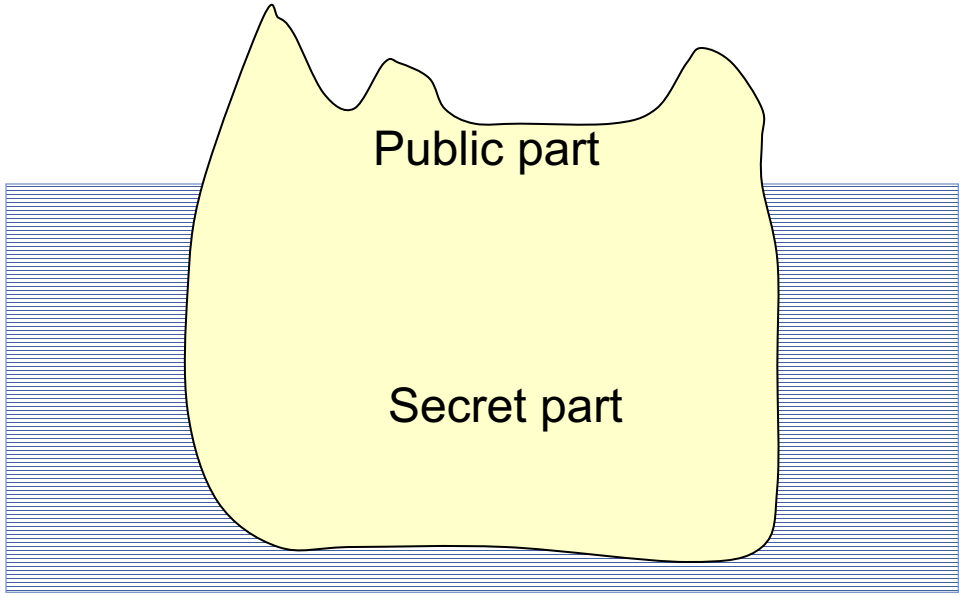
- It is part of Design
- What is the structure of the system?
  - What are the software modules (e.g. classes)?
  - What is the relationship between them (e.g. client-supplier or inheritance)?
  - What are the externally visible properties of modules?
  - API or Interface: Externally visible properties are those assumptions that other modules using it can be guaranteed e.g. services, performance etc.
- Information hiding: each module hides a design decision
  - A design decision is the the secret of the module if the design decision can be changed without affecting other modules

Architecture



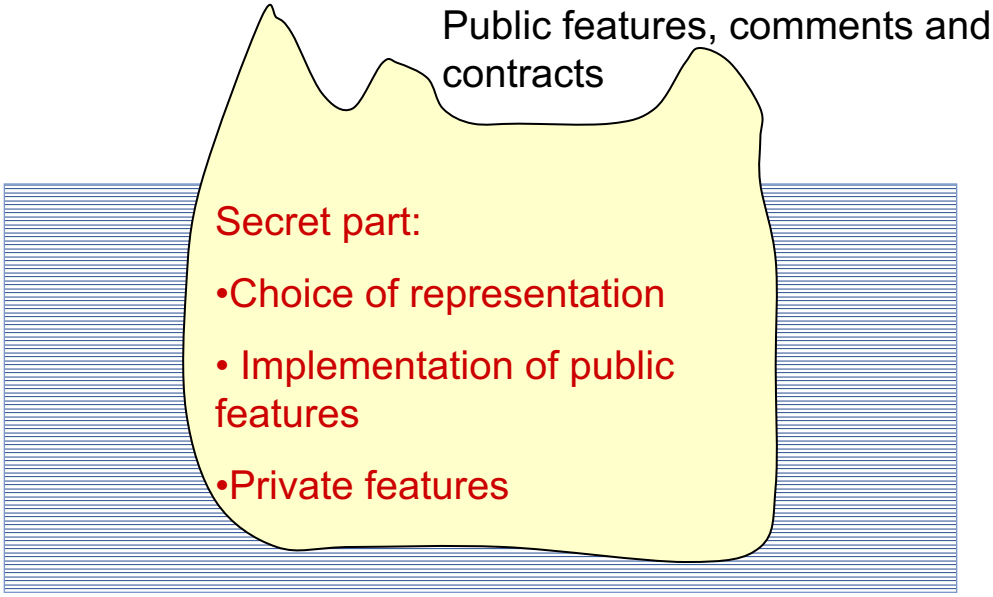
What is Information Hiding?

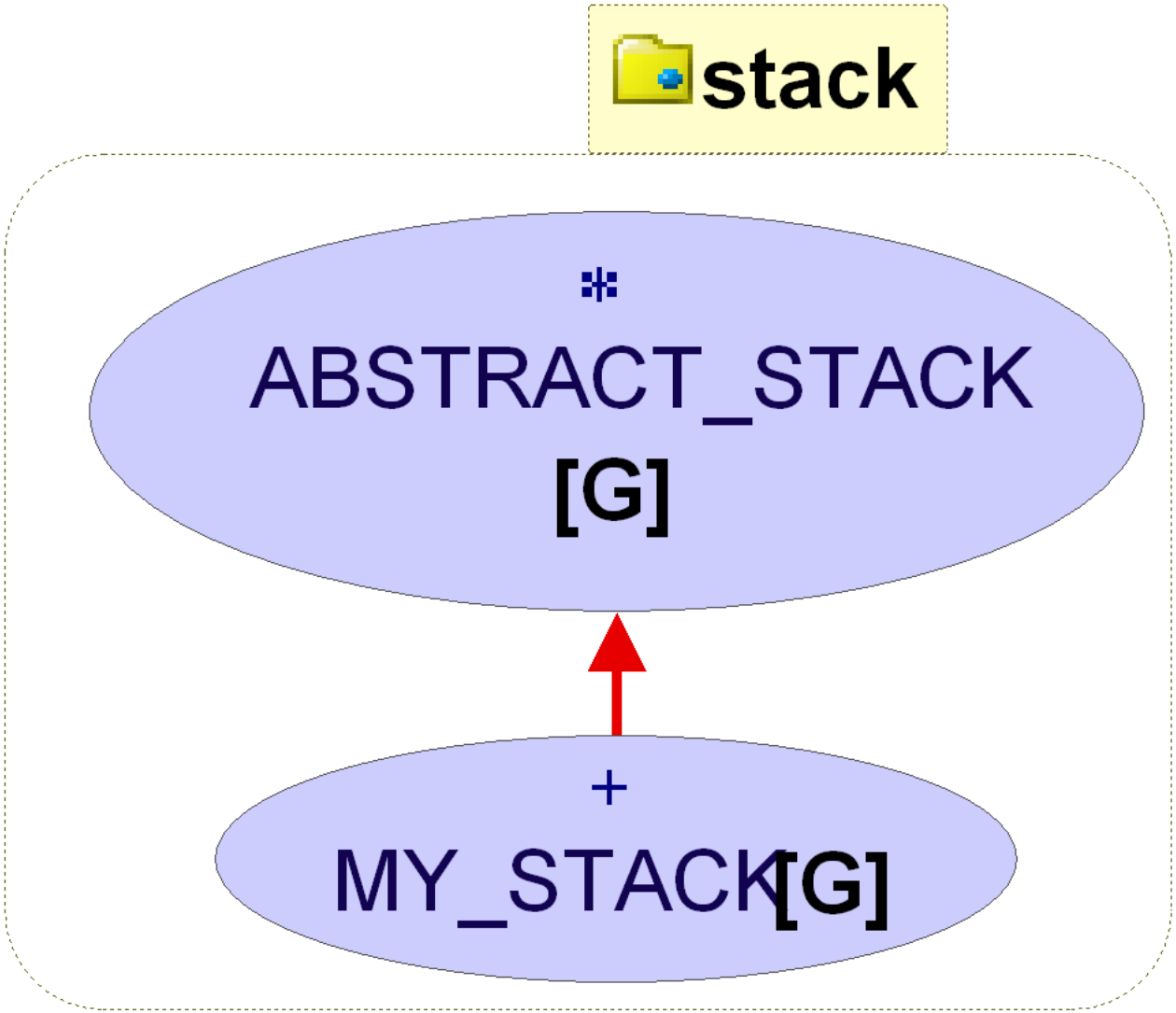
Information hiding

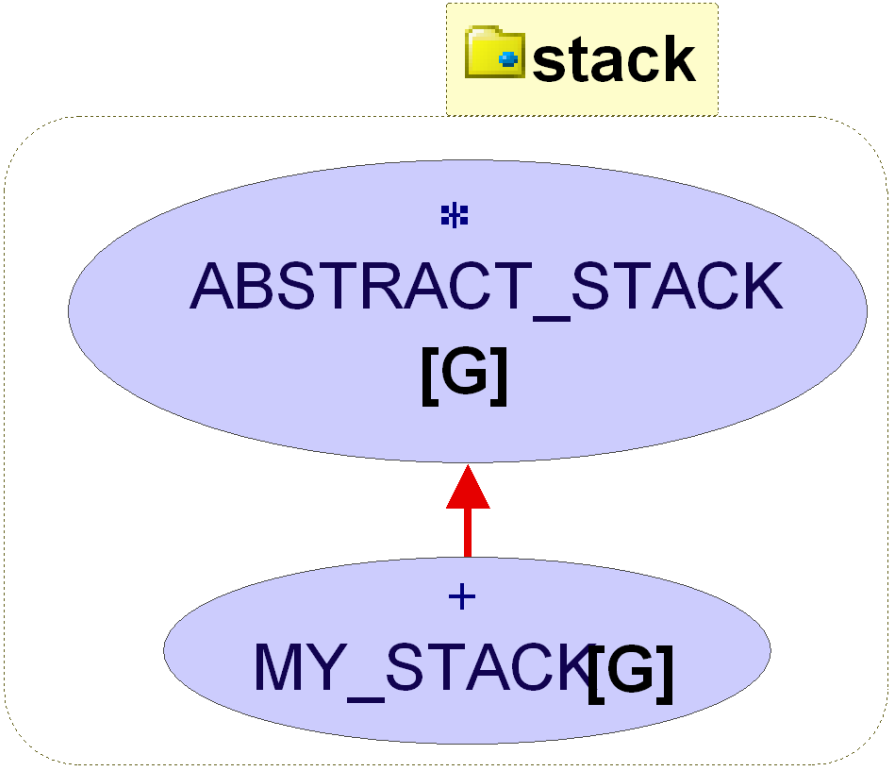


- The designer of every module must select a subset of the module's properties (public features) as the official information about the module, to be made available to authors of client modules
- Benefit: Implementation can be changed (e.g. for better efficiency) without affecting clients

Application to information hiding




stack
*
ABSTRACT_STACK
[G]
+
MY_STACK[G]



Design principle:
Code to Interface not to Implementation

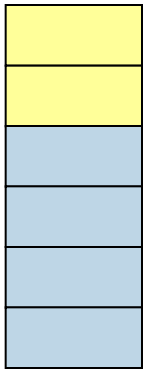
ABSTRACT_STACK

```
deferred class ABSTRACT_STACK[G]  feature
   count: INTEGER

   item: G
        --top of stack
     deferred
     end

   put (x: G)
       -- push `x' on to the stack
     deferred
     ensure
      count = old count + 1 and item = x
   end

   remove
        -- pop top of stack
     require
        count > 0 …
invariant
   count_non_negative: count >= 0
end
```

Classic Contracts

Variety of implementations

(array_up)

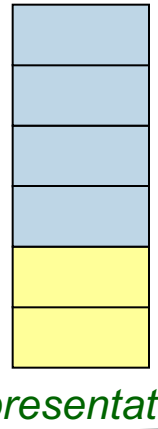


"Push" operation:

count := *count* + 1

representation [*count*] := *x*

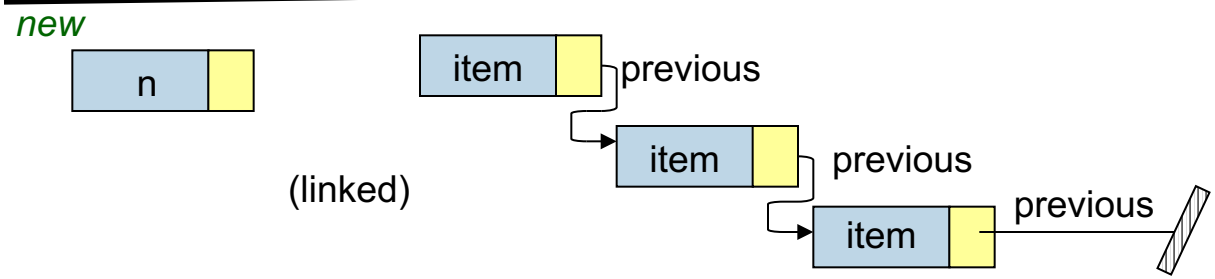
(array_down)



"Push" operation:

representation [*free*] := *x*

free := *free* - 1



"Push" operation:

new (*n*)

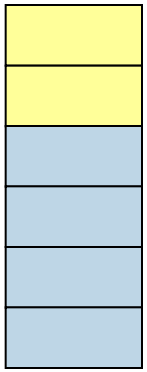
n.item := *x*

n.previous := *last*

head := *n*

Variety of implementations

(array_up)

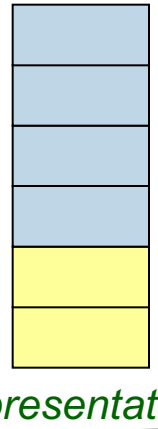


"Push" operation:

count := *count* + 1

representation [*count*] := *x*

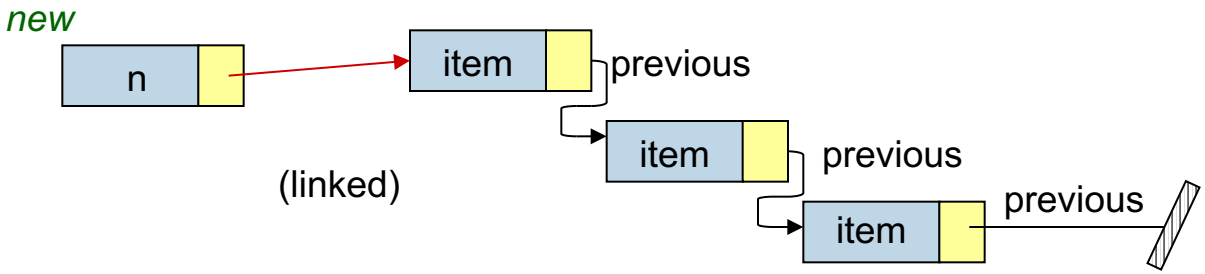
(array_down)



"Push" operation:

representation [*free*] := *x*

free := *free* - 1



"Push" operation:

new (*n*)

n.item := *x*

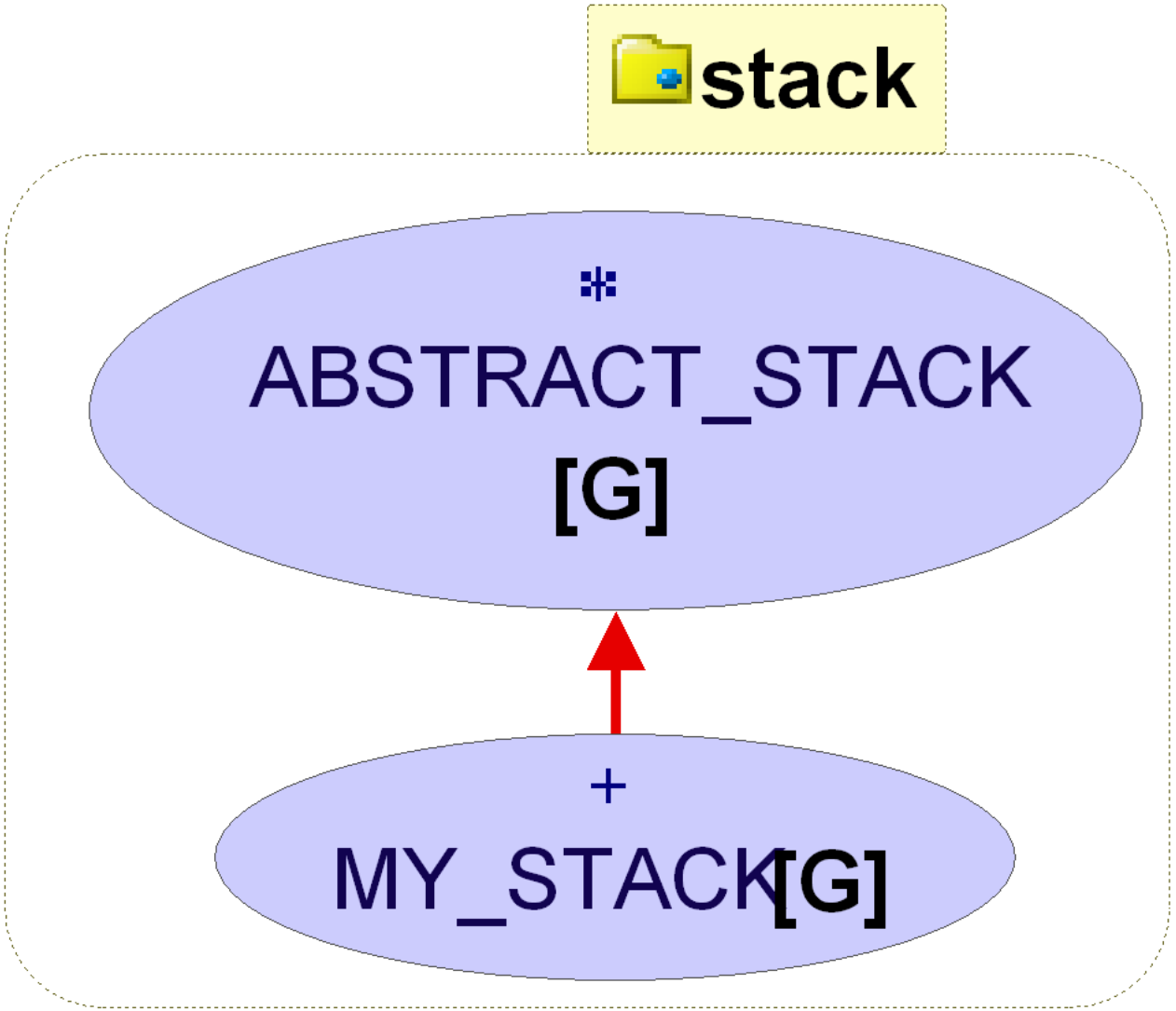
n.previous := *last*

head := *n*

Implementing STACK

```
class MY_STACK[G] inherit ABSTRACT_STACK[G] create
  make
feature
  make
    do
      create imp.make_empty
      count := 0
    end

  item: G

  count: INTEGER

  put (x: G)
     -- push `x' on to the stack
    do
      imp.extend(x)
      count := count + 1
      item := x
    ensure
     count = old count + 1 and item = x
  end

feature {NONE}
  imp: LINKED_LIST[G]
  ...
end
```

See Eiffel101 section 18.1



97

Inheritance and Information Hiding

Information hiding only applies to use by clients, using dot notation or infix notation, as with *a1.f* (*Qualified* calls)

Unqualified calls (within class) not subject to information hiding:

```
class A feature {NONE}
      h ... do  ... end
feature

      f
do
     ...; h ; ...
end
end
```

Subclasses also have access to unqualified private routines of the parent

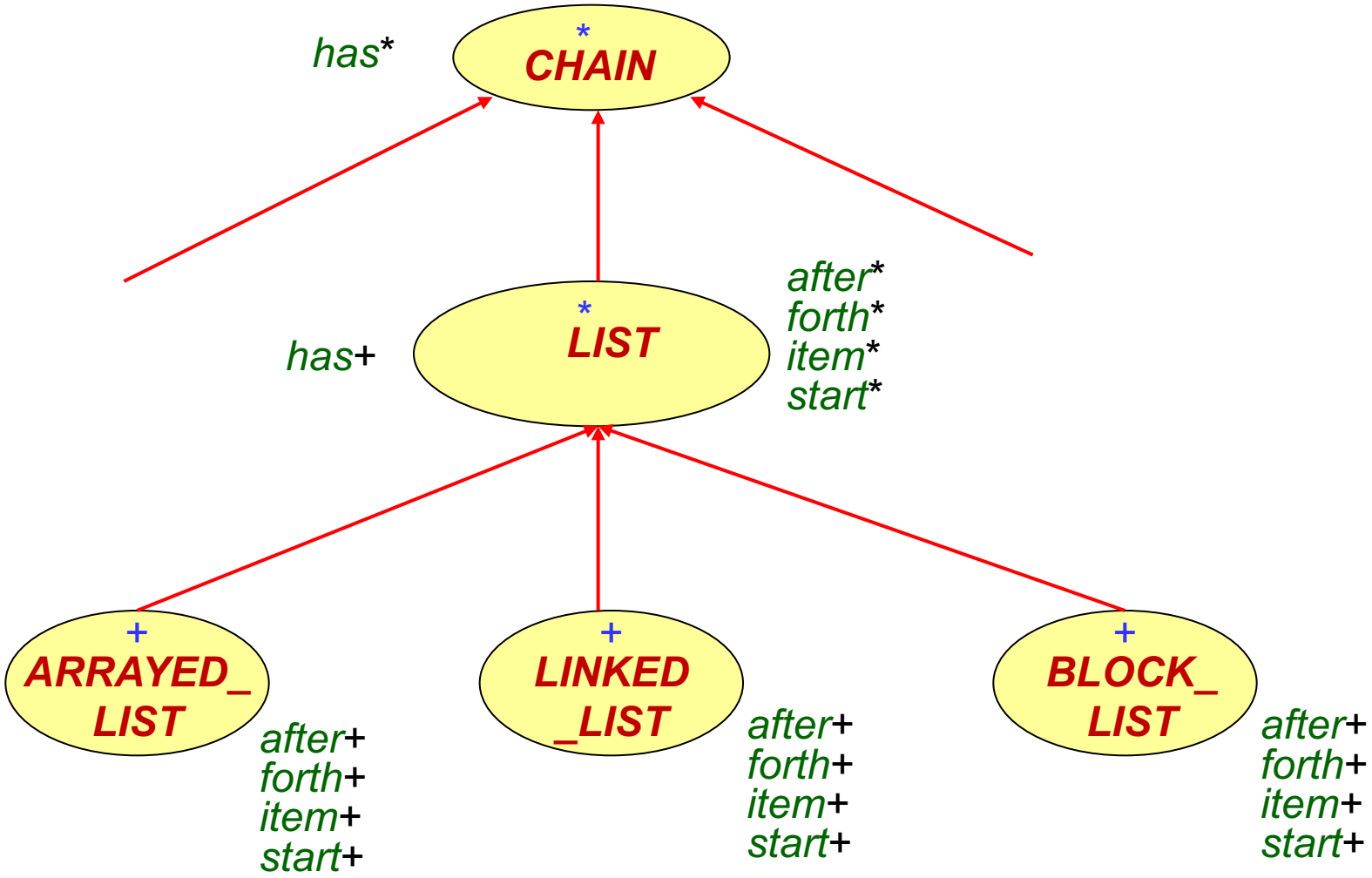
Information Hiding is for Client-Supplier Relations not Inheritance

Abstraction, information hiding, and encapsulation? What are they?

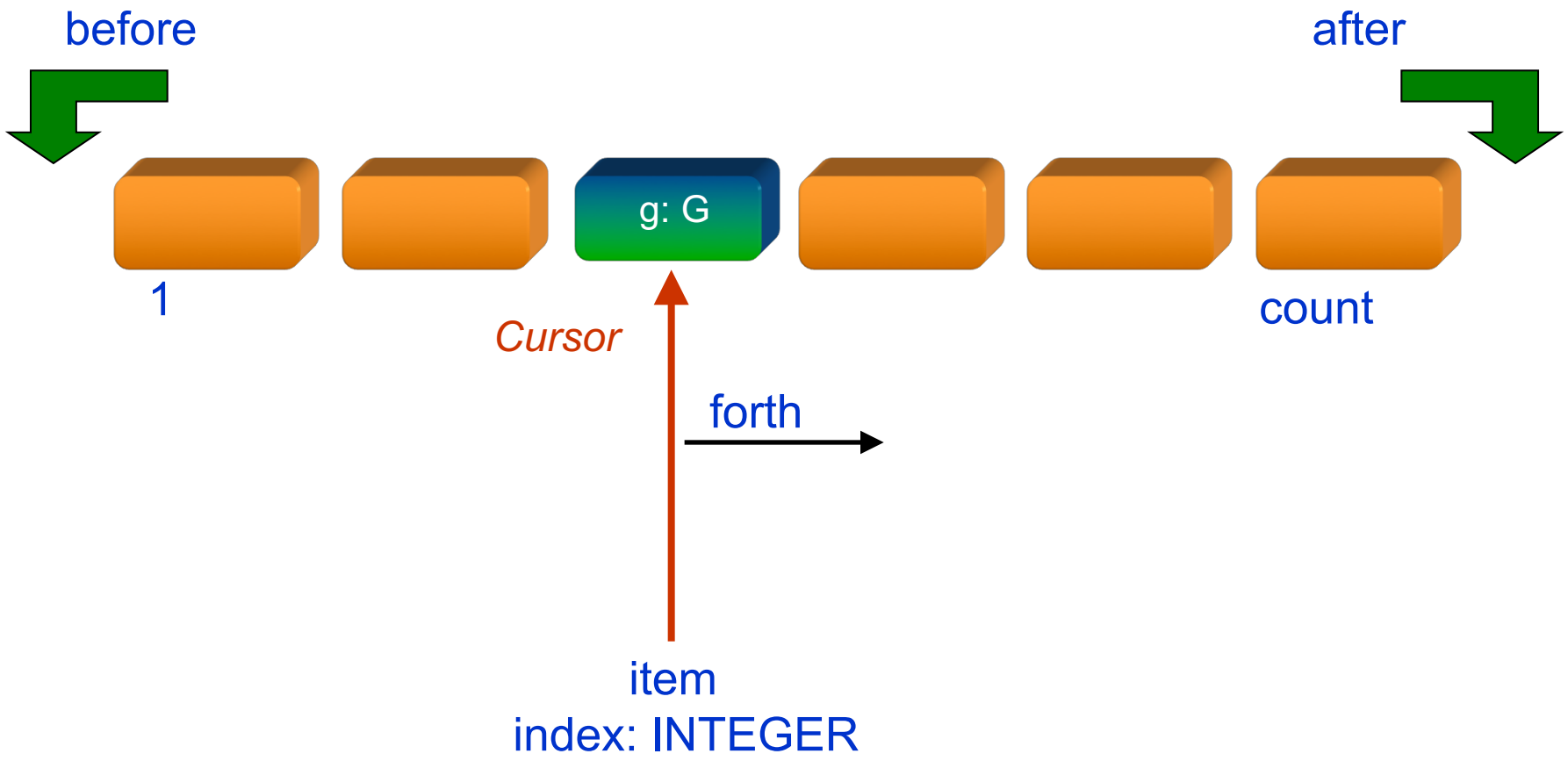
- Abstraction, information hiding, and encapsulation are very different, but highly-related, concepts
  - Abstraction is a technique that helps us identify which information should be visible, and which information should be hidden, to reduce complexity. Associated with an abstract model.
  - Encapsulation is a technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible (e.g. packaging data in a class and a sequence of statements in a routine)
  - The purpose of information hiding is to hide design decisions ("stuff" that changes) that should not affect other parts of a system

Example: LIST[G]

Descendant implementations



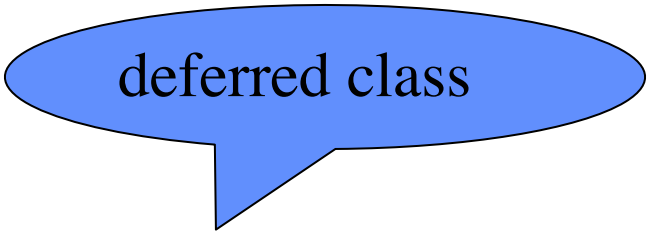
LIST [G]



LIST[G]

Generic parameter G

```
deferred class LIST[G] feature
        index: INTEGER

        after: BOOLEAN -- Valid position to the right of cursor?

        before: BOOLEAN -- Valid position to left of cursor?

        item: G
                require not off

        forth
                        -- Move to next position
                require  not_after: not after
                ensure moved_forth: index = old index + 1

        extend (v: G)
                        -- Add a new occurrence of `v'.
```

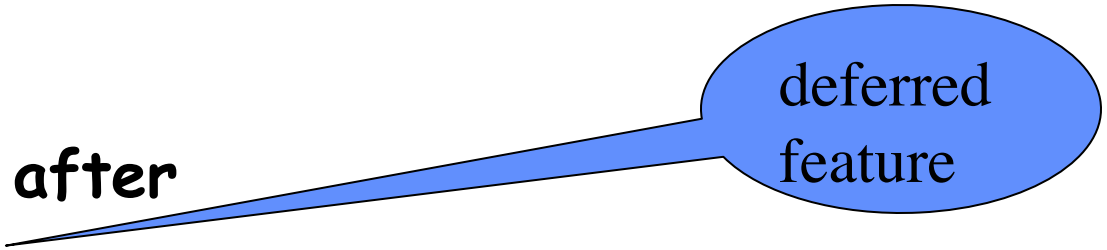


LIST[G]

```
deferred class LIST [G] inherit
        CHAIN [G]
    feature
        has (x: G): BOOLEAN is
            -- Does x appear in list?
            do
                from
                    start
                until
                    after or else found (x)
                loop
                    forth
                end
                Result := not after
        end
```

Sequential structures (cont'd)

```
forth
        require
                not after
        deferred
        ensure
                index = old index + 1
        end


start is
        deferred
        ensure
                empty or else index = 1
        end
```



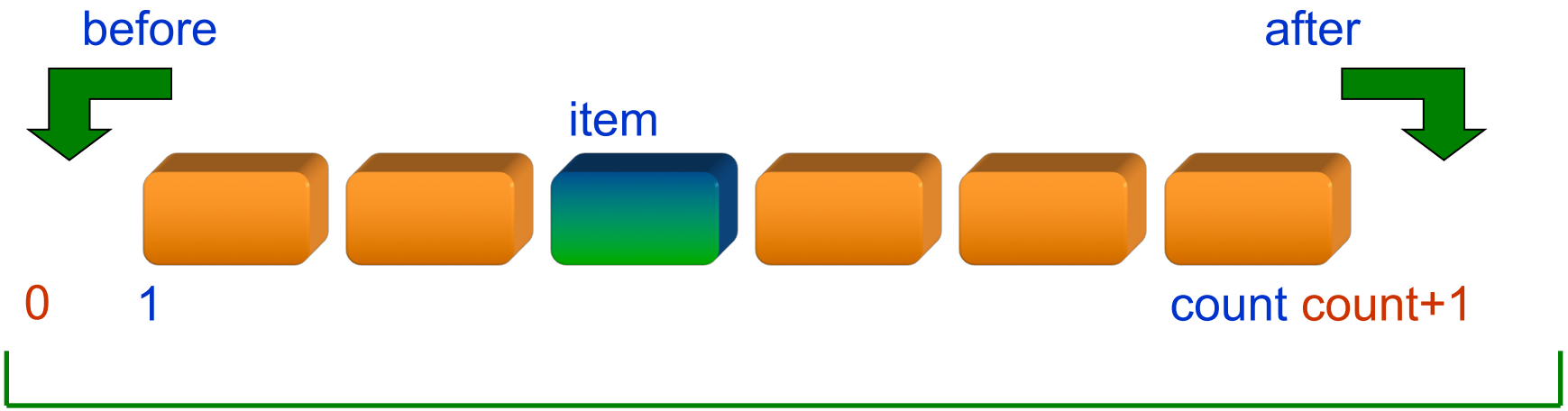
Command "forth"

Launch

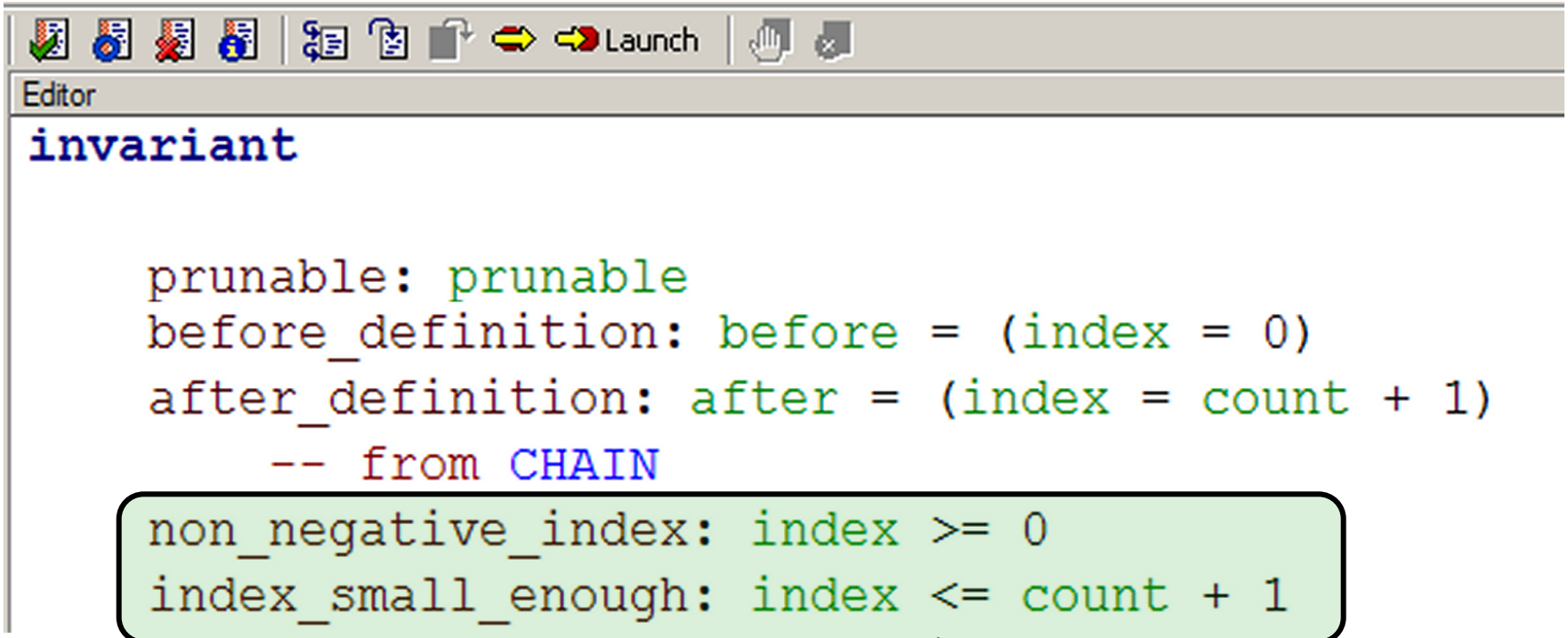
Editor

```
feature -- Status report

    after: BOOLEAN
            -- Is there no valid cursor position to the right of cursor?

    before: BOOLEAN
            -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

    forth
            -- Move to next position; if no next position,
            -- ensure that `exhausted' will be true.
        require -- from LINEAR
            not_after: not after
        ensure then
            moved_forth: index = old index + 1
```

Where the cursor may go



From the invariant of class LIST



The contract

Routine	OBLIGATIONS	BENEFITS
Client	PRECONDITION	POSTCONDITION
Supplier	POSTCONDITION	PRECONDITION

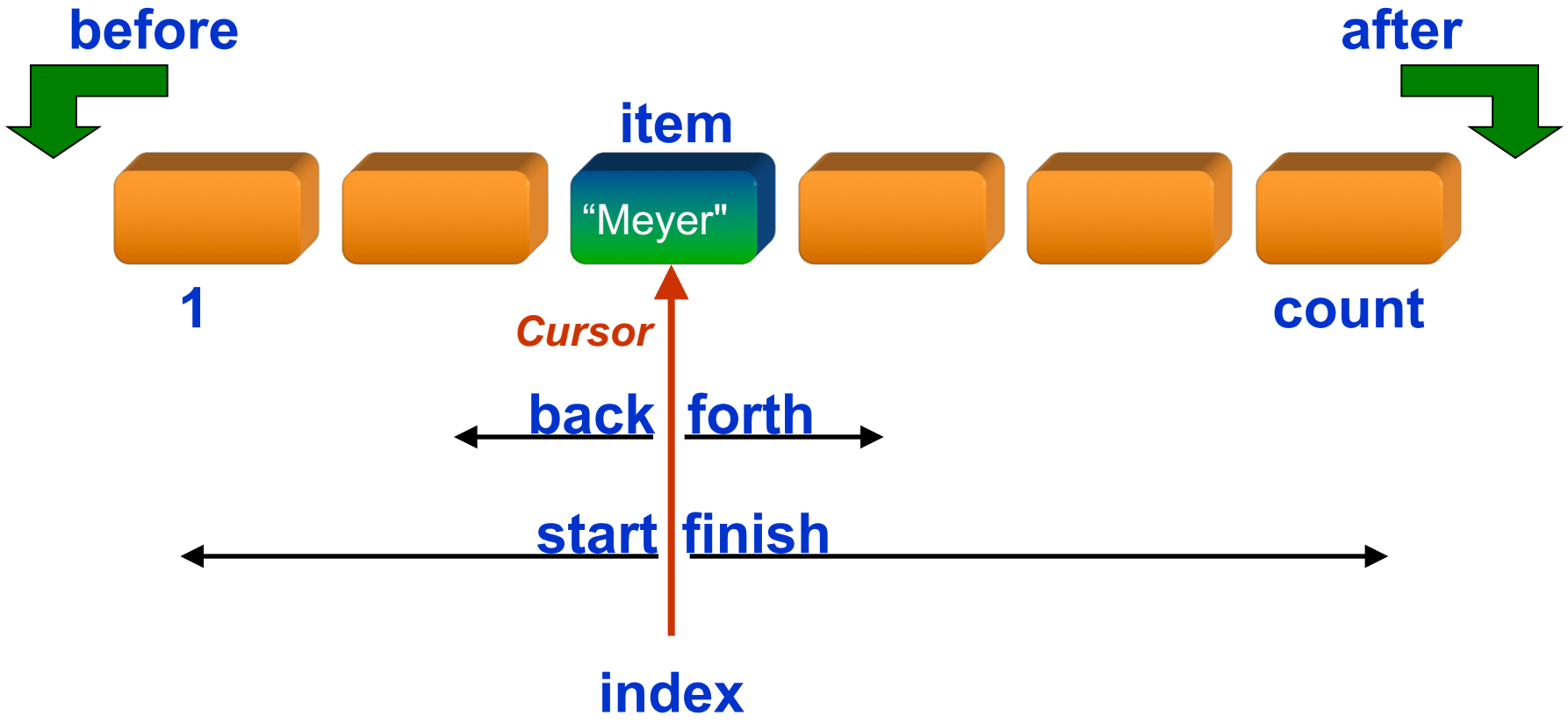
The contract of `forth'

forth	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) don't call forth when we are at the end of the list	(From postcondition:) Cursor is moved one item further
Supplier	(Satisfy postcondition:) Ensure that *item* and *index* are properly updated and consistent	(From precondition:) Don't have to handle the case where we are at the end of the list.

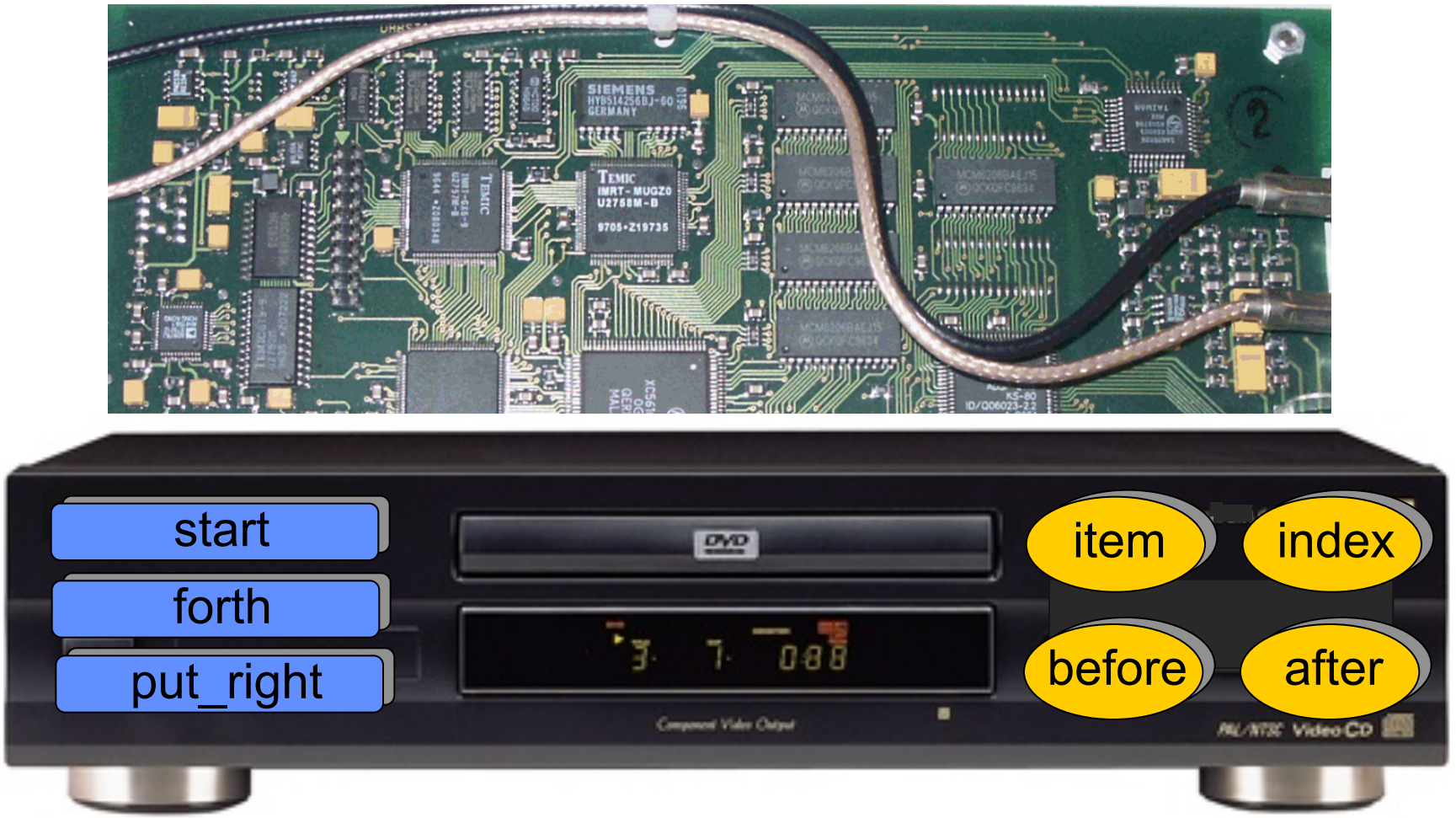
An object is a machine

An object has an interface

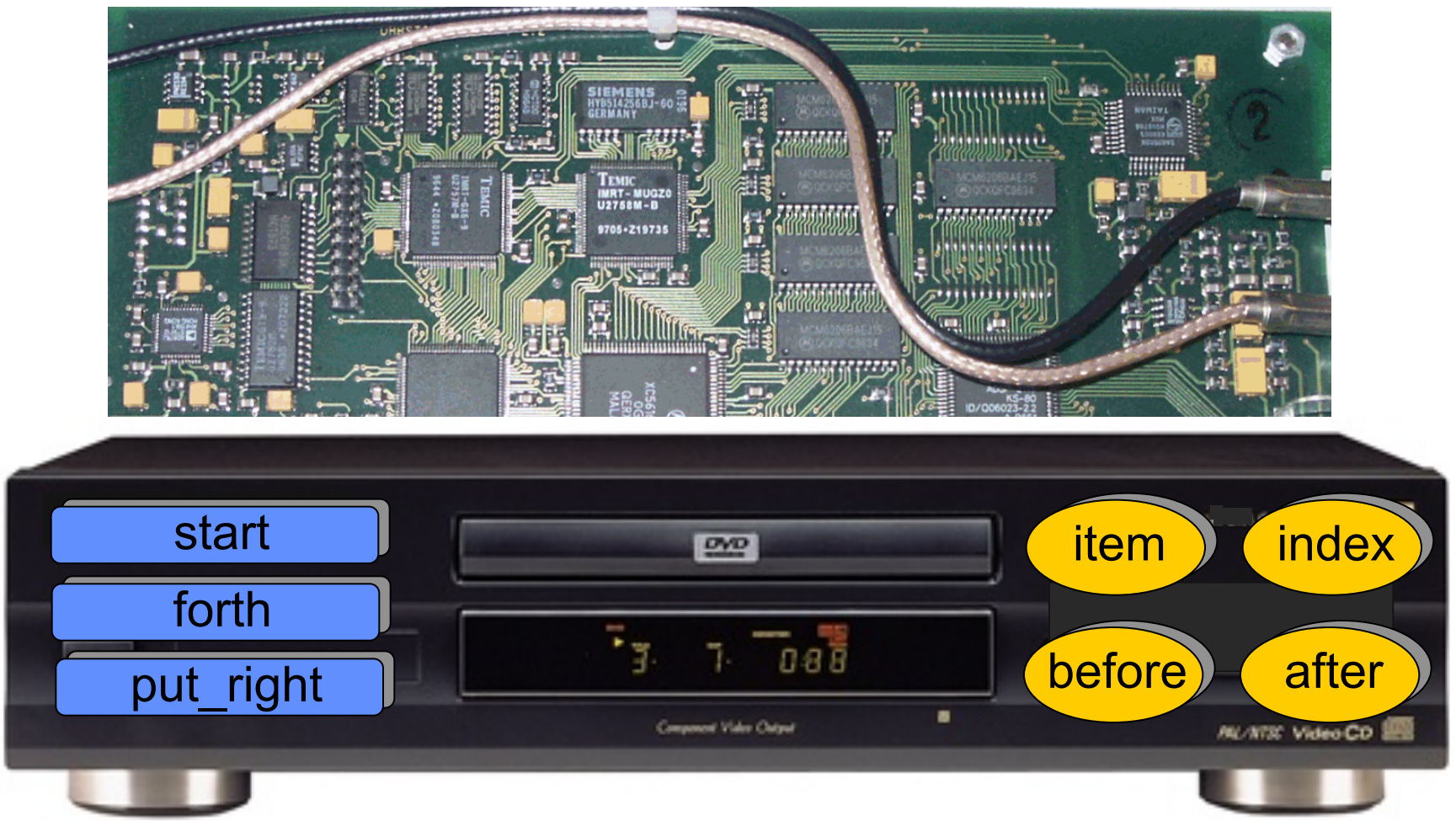
A List


before
after
item
"Meyer"
1
count
Cursor
back
forth
start
finish
index

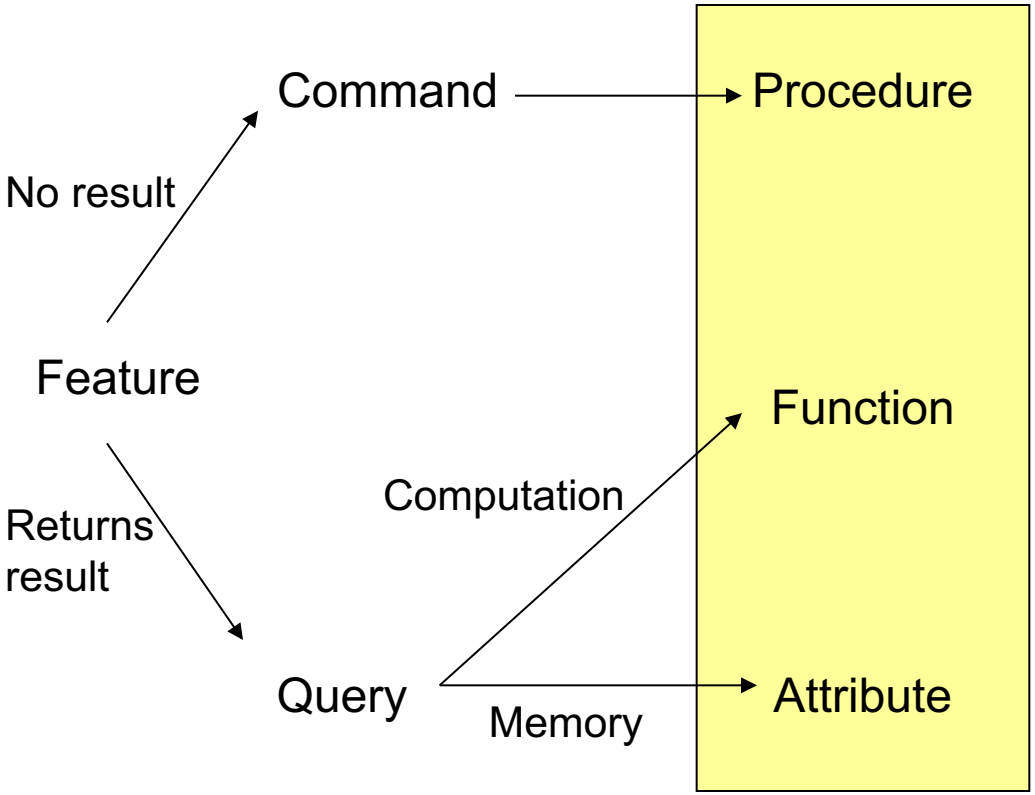
An object has an implementation



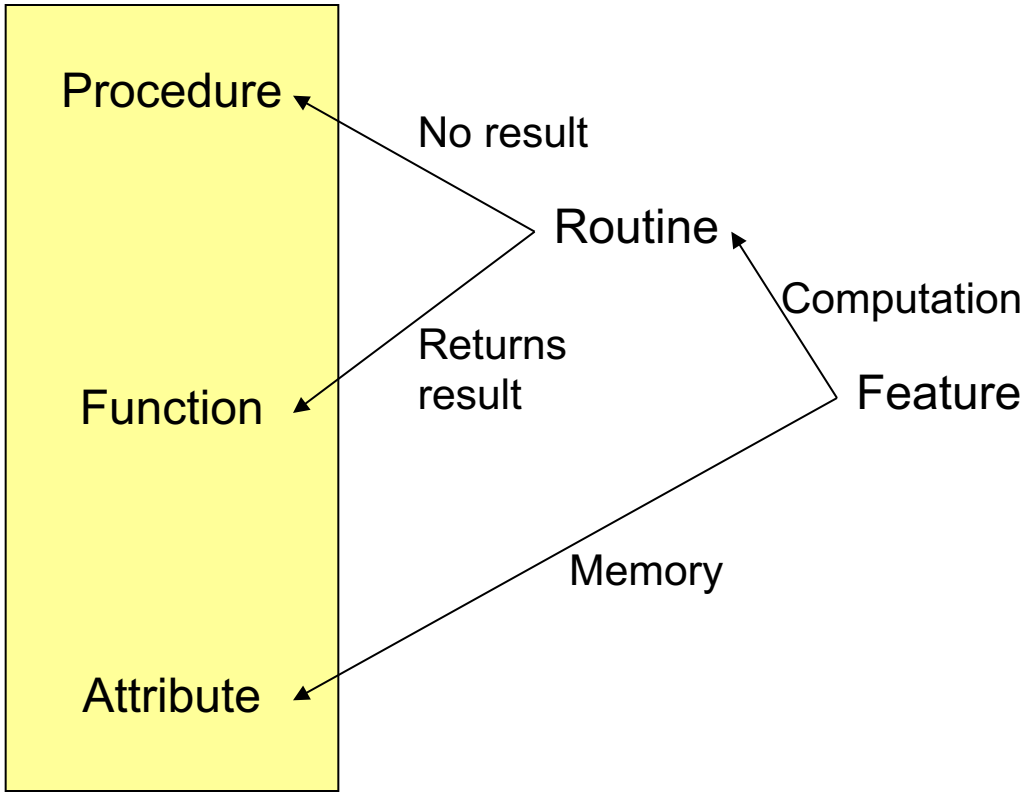
Information hiding



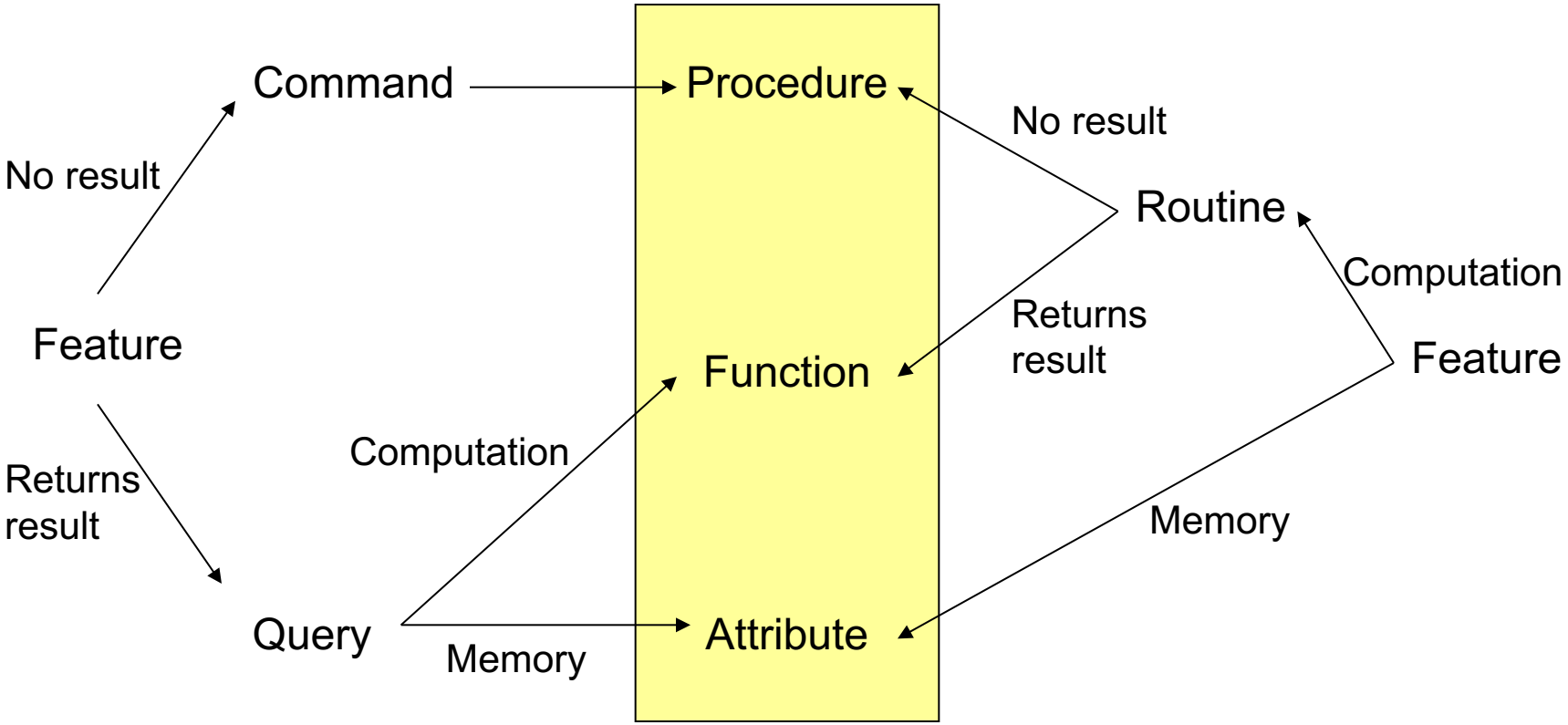
Feature categories by role



Feature categories by implementation



Feature categories



Command-Query Separation Principle

- Commands may have side effects. Queries may not.

- Why?

```
class OHM_METER
    set

feature -- comm
    set(a_curre
        require
            a_c
        do
            i := a_current
            v
        end

feature -- quer
    v: REAL  --

    i: REAL -- current in amps

    r: REAL
            -- resistance in ohms
        require
            i /= 0
        do
            i := i + 1
            Result := v/i
        end
invariant
    non_zero_current:
        i /= 0 and then r = v/i
end
```

```
t0: BOOLEAN
    local
        om: OHM_METER
    do
        comment("t0: ohms = 10volts/5amps")
        create om.set (5, 10)  -- (current, volts)
        Result := om.r = 2
        assert_equal ("test", 2, om.r)
    end
```

FAILED	Check assertion violated.	t0: ohms = 10volts/5amps Assert Equal Violation: test Expected: 2 Actual: 0.909091

Violation of command query separation principle

```
r: REAL
        -- resistance in ohms
    require
        i /= 0
    do
        i := i + 1
        Result := v/i
    ensure
        i = old i and v = old v
    end
```

FAILED	Postcondition violated.	t0: ohms = 10volts/5amps

- In implementation languages such as Java, how to avoid unacceptable/infinite outputs?
- Place an assert before/after each method? How to ensure that future methods do not produce unacceptable outputs? (need invariants)
- How do you assign which module is to blame when things go wrong? Client or supplier?
- How do you design your code with the aim of correctness, rather than ad-hoc (hacking?)
- Can a design be good if it is incorrect?
- How can we determine if the design is correct?

```
class OHM_METER create
    set

feature -- commands
    set(a_current, a_voltage:REAL)
        require
            a_current /= 0
        do
            i := a_current
            v := a_voltage
        end

feature -- queries
    v: REAL  -- volts

    i: REAL -- current in amps

    r: REAL
            -- resistance in ohms
        require
            i /= 0
        do
            i := i + 1
            Result := v/i
        end
invariant
    non_zero_current:
        i /= 0 and then r = v/i
end
```

Equipping any class with array notation

```
class LIST[G] feature
   item: G
   i_th alias "[]" (i: INTEGER): like item assign put_i_th
   put_i_th (g: G; i: INTEGER):
        require valid_index (i)
```

```
juliet, romeo: STRING;      people: LIST[STRING]
check people[2] = juliet end
   -- syntactic sugar for people.i_th(2) = juliet
people[2] := romeo
   -- syntactic sugar for population.put_i_th(romeo,2)
```

Is the following a good design?

```
class PERSONNEL_DATA feature
  names: ARRAY[STRING]     --employee name
  addresses: ARRAY[STRING]
  phones: ARRAY[INTEGER]
  supervisor_names: ARRAY[STRING] – supervisor data
  titles: ARRAY[STRING]
  bonuses: ARRAY[VALUE]

  extend(n,a: STRING;p:INTEGER; supervisor: STRING)
        -- add new person with their supervisor to the database
end
```

name	address	phone
Pam	100 Keele	736-2100
Tom	20 Younge	710-4556

Some take home lessons

- The importance of Requirements – happy customers
- Specifications (contracts) vs. Programs
- Design by Contract (DbC)
- Code to Interface not to Implementation
- What is Design?
- Information Hiding Principle (hide design decisions)
- Uniform Access Principle
- Test Driven Development (TDD)
  - Unit tests and debugging
- Reference vs. Object Comparisons
- Generic Parameters
- BON/UML class diagrams

Professional Software Development

- Develop software that is
  - Correct
  - Efficient
  - Maintainable (well designed, thus easy to read, thus easy to fix and modify)
  - On time and within budget

https://www.codethink.co.uk/join-us.html

Systems and Software Engineer

As a Codethink Systems and Software Engineer you can expect to work on a variety of challenging software projects servicing our clients across many different industries. Within this role you will be expected to:

- Participate in all phases of the full software development lifecycle
- Write well designed, testable, efficient code
- Produce specifications and determine operational feasibility
- Integrate software components into a fully functional software system
- Develop software test and verification plans and procedures
- Tailor and deploy software tools, processes and metrics
- Follow project plans and industry standards

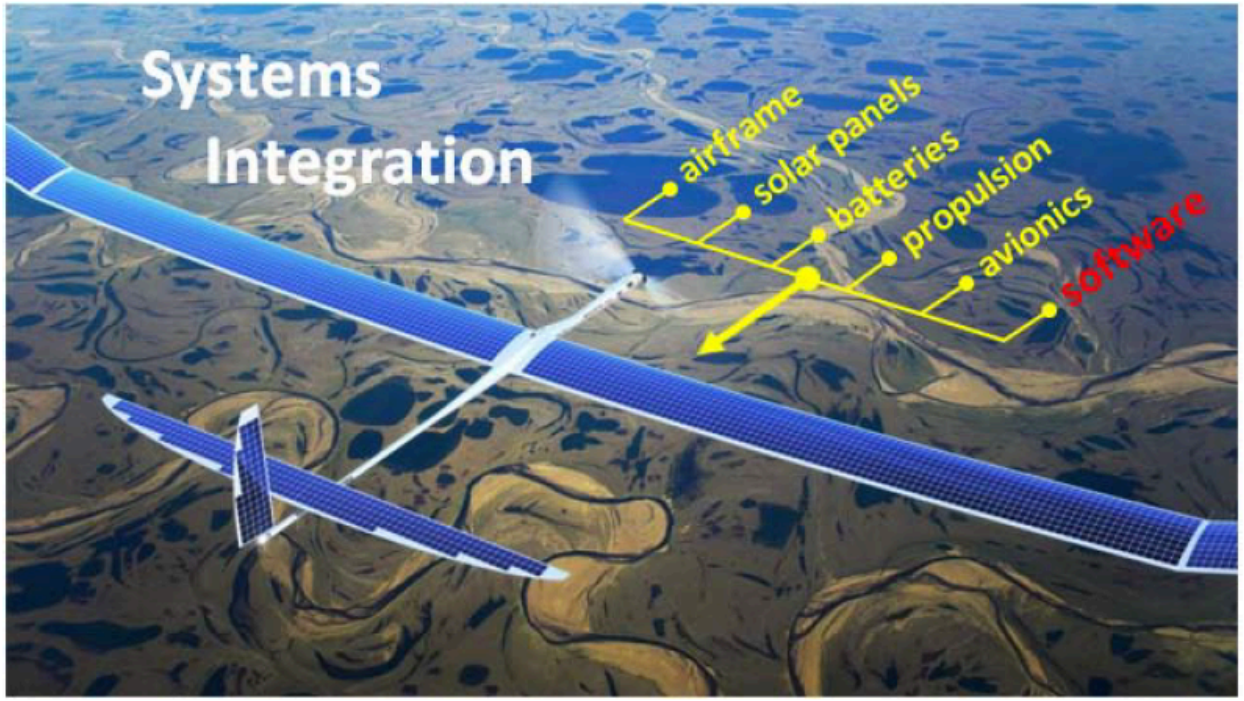
Skills & requirements:

- All applicants should be/become comfortable using a linux desktop and working with free and open source software.
- Must be able to demonstrate intellect and intelligence, via academic achievement and/or other means.
- Must be capable of dealing with pressure, uncertainty and deadlines.
- Must demonstrate both willingness and ability to grasp new technical concepts.
- Key technologies we work with: Any of Linux, C/ C++, C#, Python, Java, device drivers, embedded

Very little software exists in isolation these days, it is often an integral part of a larger system involving many technologies. Titan Aerospace's Solara is a good example. Solara is a solar powered drone aircraft designed to deliver satellite services without leaving the atmosphere (refer Figure 9). This air vehicle exists because a market for cheaper communication systems has coincided with advances in solar panel efficiency, battery chemistry, materials science and airframe design. Software is only part of the story.

Building this system requires a variety of disciplines in addition to software engineering. This is a classical systems integration project where effective interaction between disciplines produces the collective competence that solves wicked problems.

Figure 9. Solara, a solar powered drone, integrates software with multiple technologies



A software developer with knowledge of related engineering disciplines is highly valued in projects such as this. At the very least, developers need to accept other cultures, forgive them for their lack of software knowledge, spend time educating them on what they need to know and discover ways of working.