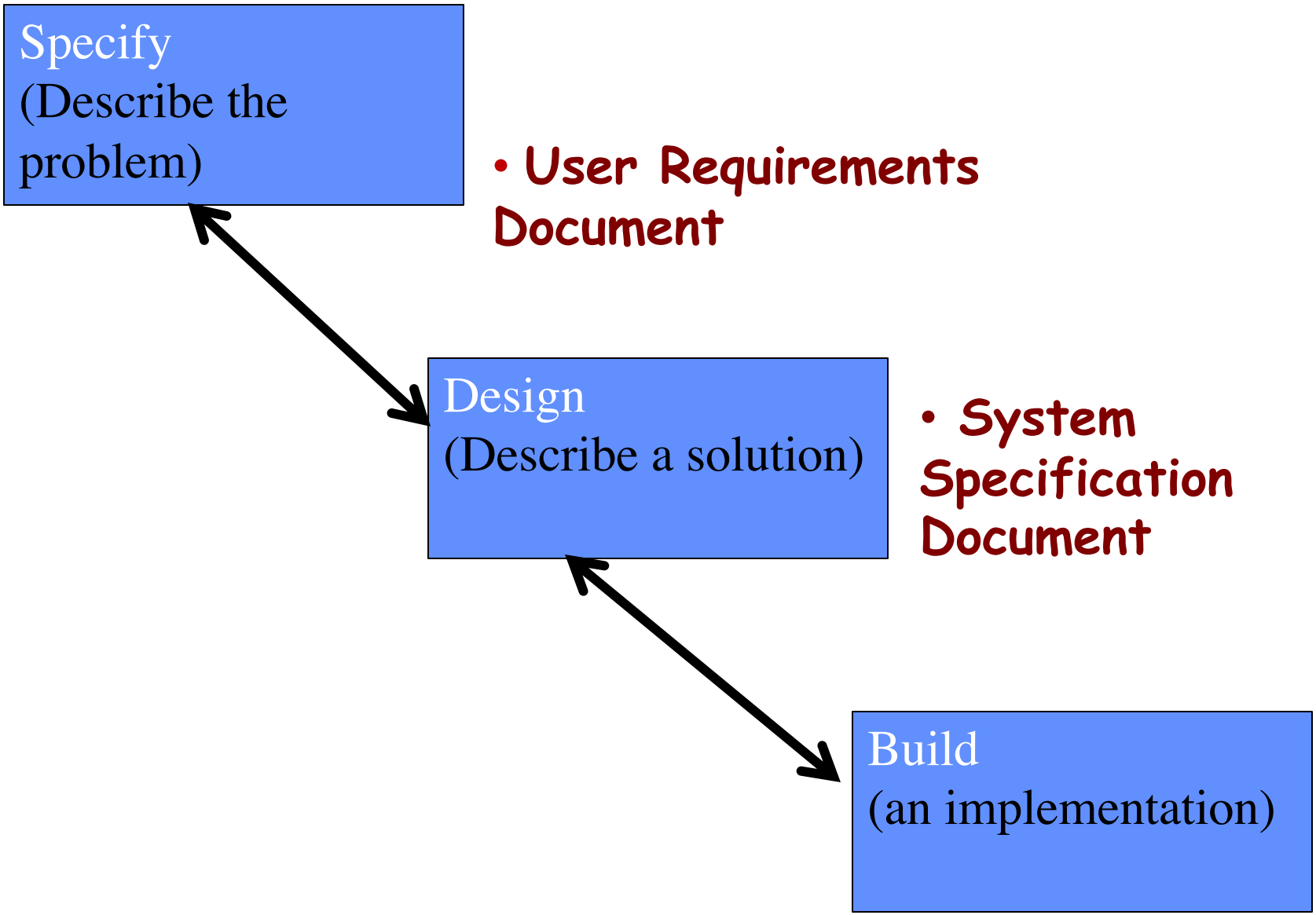
YORK
UNIVERSITÉ
UNIVERSITY

Software Design
CSE3311

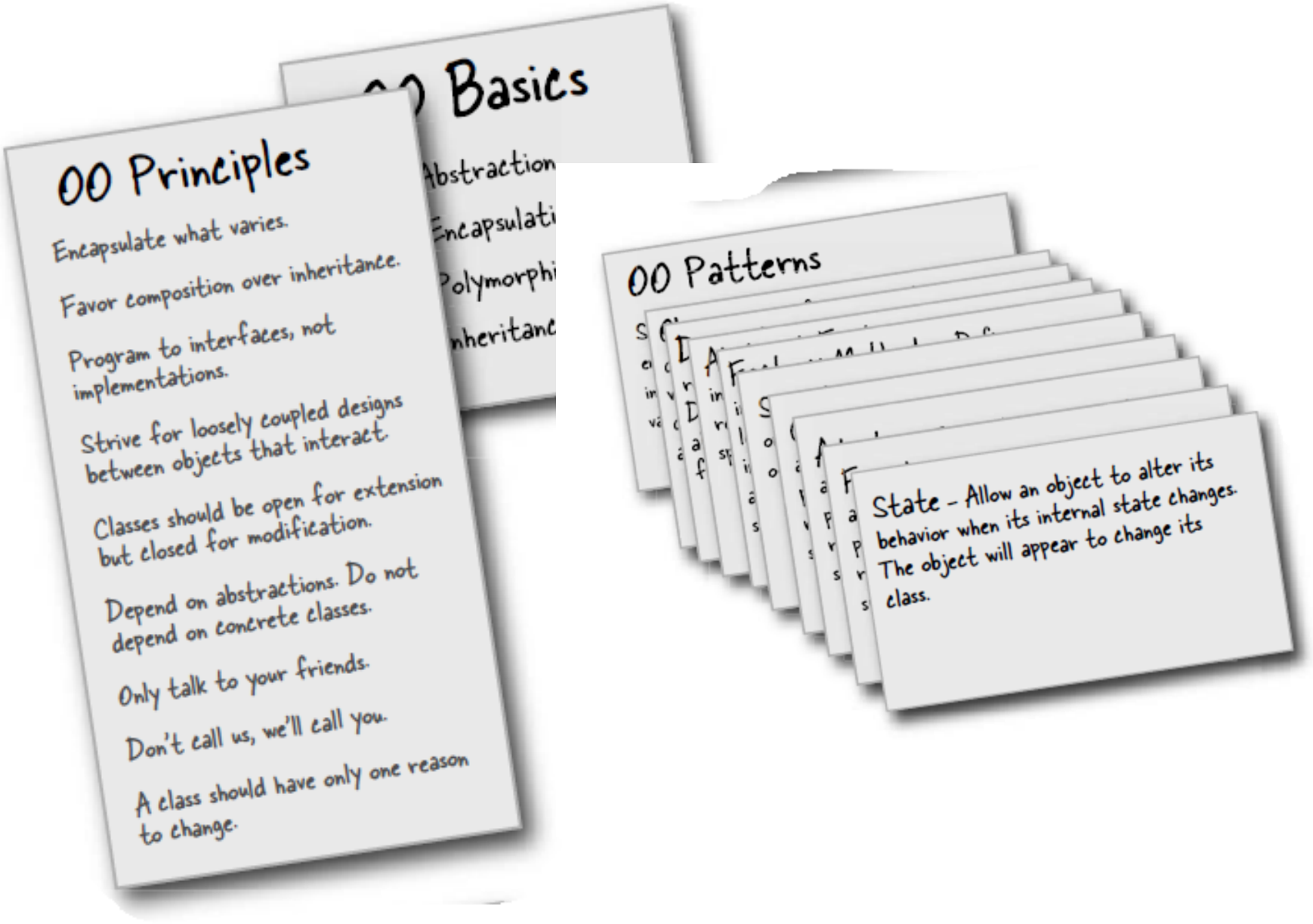
Design Patterns

Iterator Pattern

Singleton Pattern


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

General OO Design Principles



What is a Design Pattern?

- *A* **pattern**
  - Describes a problem which occurs repeatedly in the design of different software products
  - Describes the core of a solution
  - The core description is re-usable

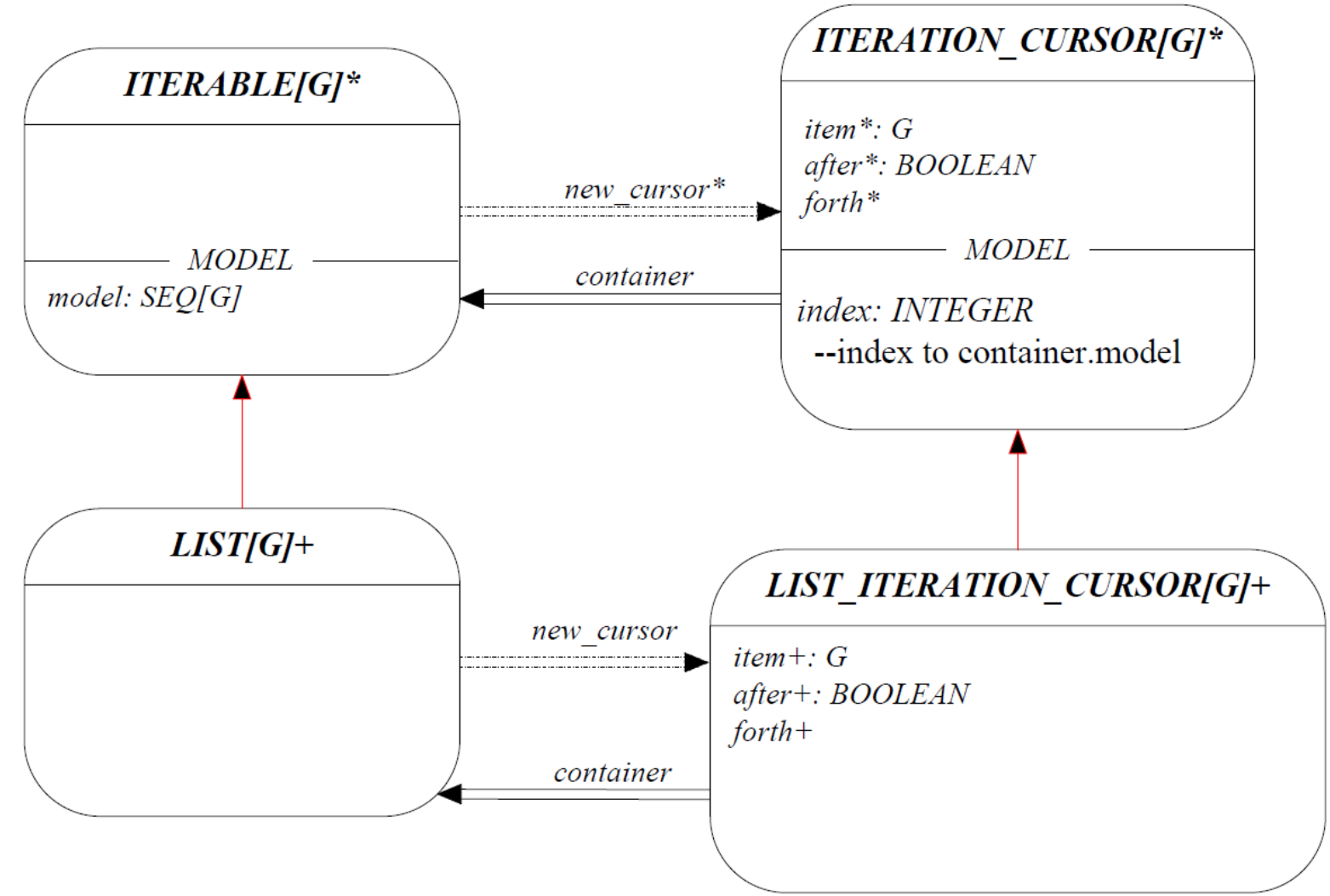
Iterator Design Pattern

- **Problem:**
  - There exists a need to abstract the traversal of widely different data structures (arrays, lists, sets, maps, etc.) so that algorithms can be defined that are capable of interfacing with each data structure transparently.
- **Intent:**
  - Provide a way to access the elements of an aggregate object sequentially (via a "cursor") without exposing its underlying representation

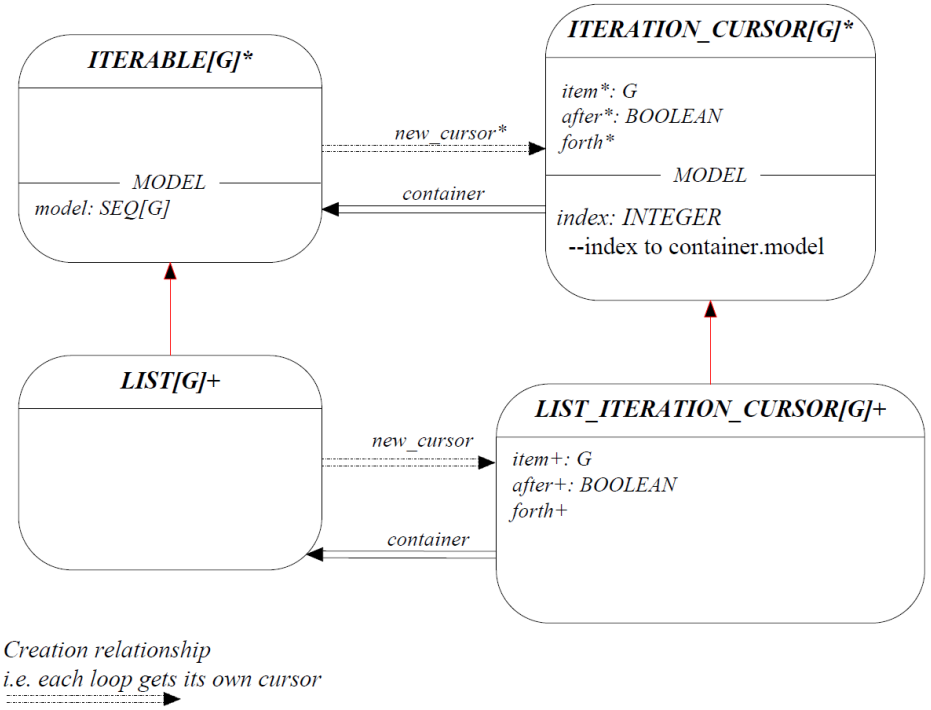
See Eiffel Snippets

- svn export https://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator
  - Login anonymous, no password
- http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator/index.html

Iterator Design Pattern



across Construct




```
class CLIENT feature
  collection: LIST[STRING]

  print_collection
    local
      cursor: ITERATION_CURSOR[STRING]
    do
      from
        cursor := collection.new_cursor
      until
        cursor.after
      loop
        print(cursor.item)
        cursor.forth
       end

  end
```

```
print_collection
  do
    across
      collection
    as
      cursor
    loop
      print(cursor.item)
    end
  end
```

new_cursor

- In class LIST[G] implement new_cursor

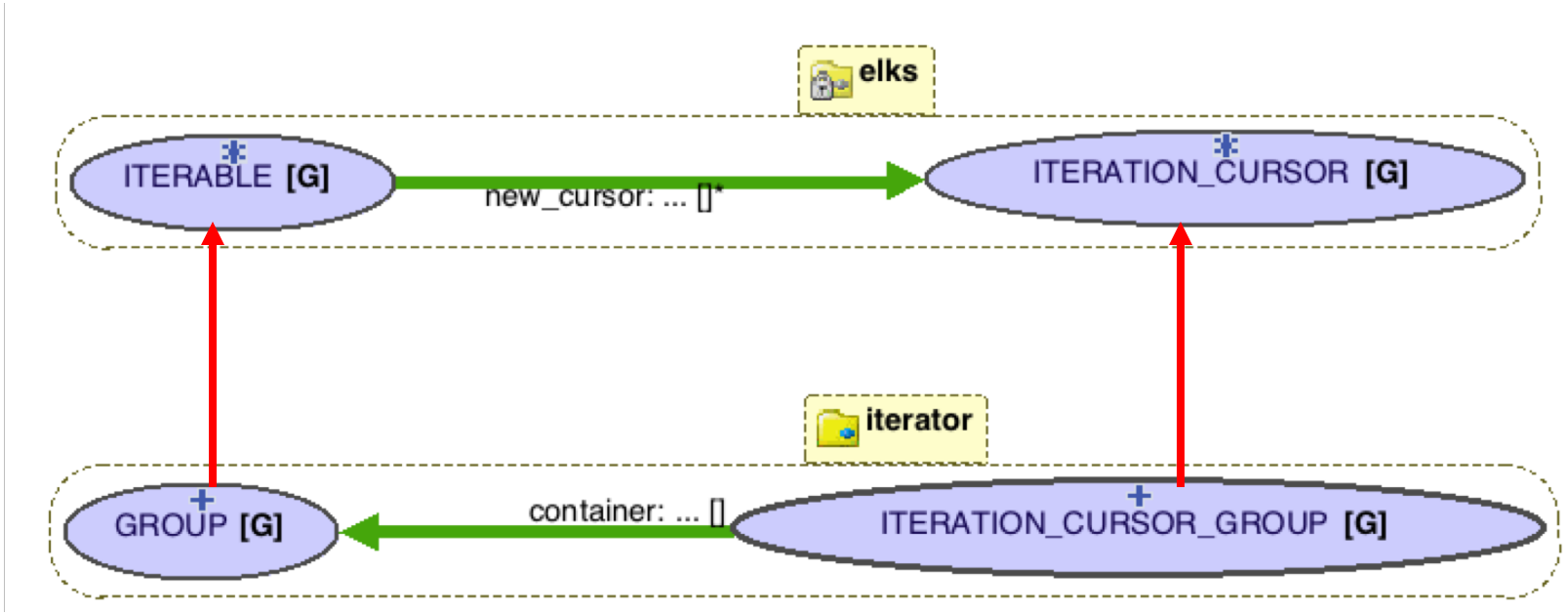
```
new_cursor: LIST_ITERATION_CURSOR[G]
        do
                create Result.make(Current)
                Result.start
        end
```

- In class LIST_ITERATION_CURSOR[G] implement

  make, item, after, forth

Simple Example: equip class with iterator

- https://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/iterator



class GROUP[G -> attached ANY] …

```
class
    GROUP[G]
inherit
    ITERABLE[G]
        undefine
            out
        end

    DEBUG_OUTPUT
        redefine
            out
        end
create
    make
feature{NONE}
    make(a_x,a_y,a_z:G)
        do
            x := a_x
            y := a_y
            z := a_z
        end

feature
    x,y,z: G

feature -- iterable
    new_cursor: ITERATION_CURSOR [G]
        do
            create {ITERATION_CURSOR_GROUP[G]}Result.make (Current)
        end
```

```
feature -- out
    out: STRING
        do
            if attached x as xa
            and attached y as ya
            and attached z as za then
                Result := "("
                + xa.out
                + ","
                + ya.out
                + ","
                + za.out
                + ")"
            else
                Result := ""
            end

        end

    debug_output: STRING
        do
            Result := out
        end

end
```

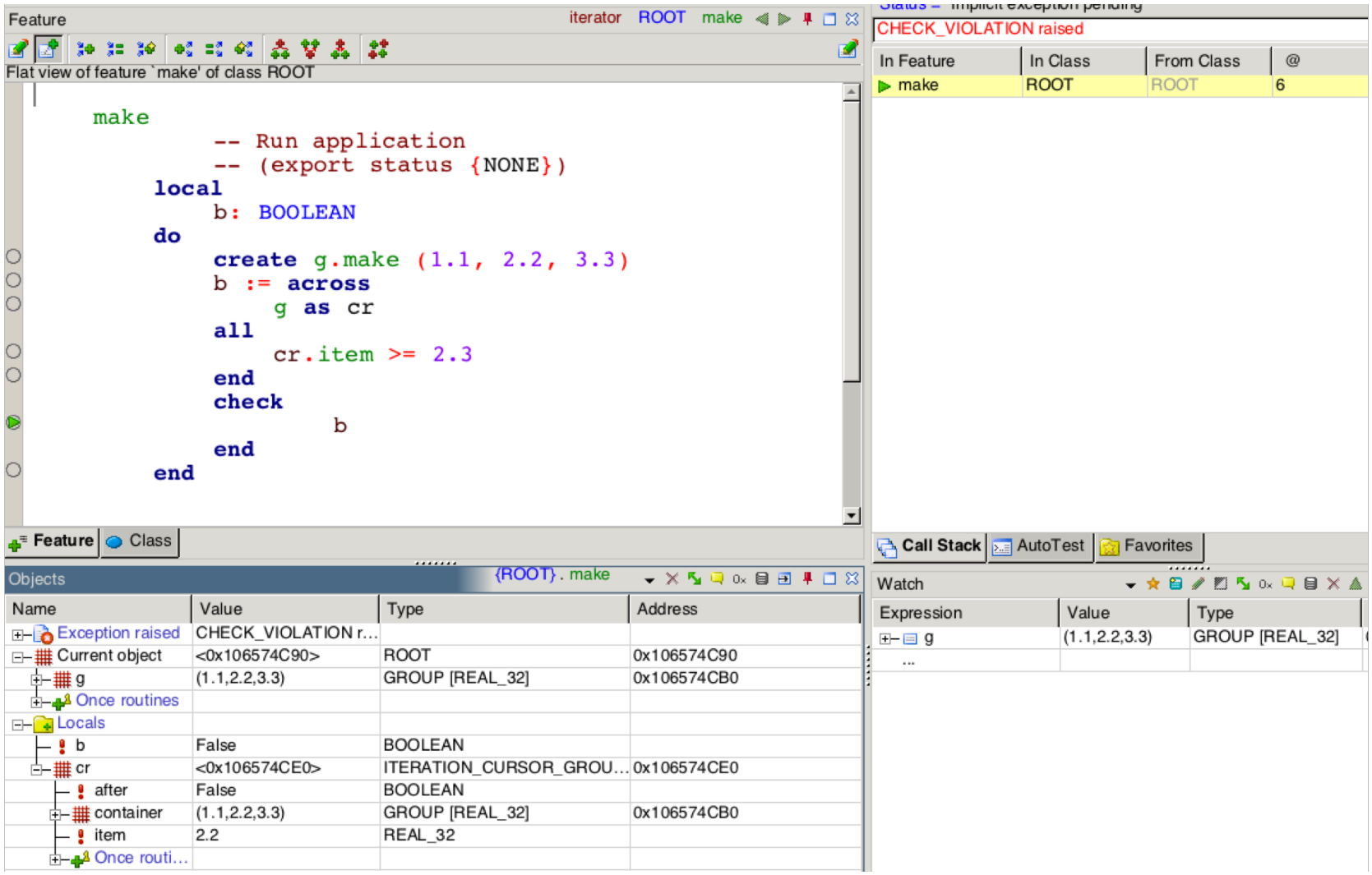
```
class GROUP [G -> attached ANY] inherit
    ITERABLE [G]
        undefine
            out
        end
    DEBUG_OUTPUT
        redefine
            out
        end
create
    make
feature {NONE}
    make (a_x, a_y, a_z: G)
        do
            x := a_x
            y := a_y
            z := a_z
        end
feature  -- items over which to iterates
    x, y, z: G

feature -- iterable
    new_cursor: ITERATION_CURSOR [G]
        do
            create {ITERATION_CURSOR_GROUP [G]}
                Result.make (Current)
        end
```

```
feature -- out
    out: STRING
        do
            Result := "(" +
                x.out + "," +
                y.out + "," +
                z.out + ")"
        end

    debug_output: STRING
        do
            Result := out
        end
end
```

```
class
    ITERATION_CURSOR_GROUP[G -> attached ANY]
inherit
    ITERATION_CURSOR[G]
create
    make
feature {NONE}
    make(a_g: GROUP[G])
            -- an instance of container
        do
            container := a_g
            item := container.x
        end

    container: GROUP[G]

feature -- Access

    item: G

    after: BOOLEAN

    forth
        do
            if item ~ container.x then
                item := container.y
            elseif item ~ container.y then
                item := container.z
            else
                item := container.z
                after := True
            end
        end
end
```

```
class ROOT create
    make

feature {NONE} -- constructor

    g: GROUP[REAL]

    make
            -- Run application
        local
            b: BOOLEAN
        do
            print("%NStart iterator tests ...%N")
            create g.make(1.1, 2.2, 3.3)

            -- all
            b := across g as cr all cr.item >= 2.3 end
            check not b end -- should succeed

            --some
            b := across g as cr some cr.item >= 2.3 end
            check b end

            -- loop
            across g as cr loop
                print(cr.item.out)
                io.new_line
            end
            print("Iterator tests succeeded")
        end
end
```

Debugger


Feature
iterator ROOT make
Flat view of feature `make' of class ROOT
make
-- Run application
-- (export status {NONE})
local
b: BOOLEAN
do
create g.make (1.1, 2.2, 3.3)
b := across
g as cr
all
cr.item >= 2.3
end
check
b
end
end
Feature
Class
CHECK_VIOLATION raised
In Feature
In Class
From Class
@
make
ROOT
ROOT
6
Call Stack
AutoTest
Favorites
Objects
{ROOT}.make
Name
Value
Type
Address
Exception raised
CHECK_VIOLATION r...
Current object
<0x106574C90>
ROOT
0x106574C90
g
(1.1,2.2,3.3)
GROUP [REAL_32]
0x106574CB0
Once routines
Locals
b
False
BOOLEAN
cr
<0x106574CE0>
ITERATION_CURSOR_GROU...
0x106574CE0
after
False
BOOLEAN
container
(1.1,2.2,3.3)
GROUP [REAL_32]
0x106574CB0
item
2.2
REAL_32
Once routi...
Watch
Expression
Value
Type
g
(1.1,2.2,3.3)
GROUP [REAL_32]
...

```
class ROOT create
    make

feature {NONE} -- constructo

    g: GROUP[REAL]

    make
            -- Run applicati
        local
            b: BOOLEAN
        do
            print("%NStart iterator tests ...%N")
            create g.make(1.1, 2.2, 3.3)
            b := across g as cr all cr.item >= 2.3 end
            check not b end -- should succeed
            b := across g as cr some cr.item >= 2.3 end
            check b end
            across g as cr loop
                print(cr.item.out)
                io.new_line
            end
            print("Iterator tests succeeded")
        end
end
```

```
Start iterator tests ...
1.1
2.2
3.3
Iterator tests succeeded
```

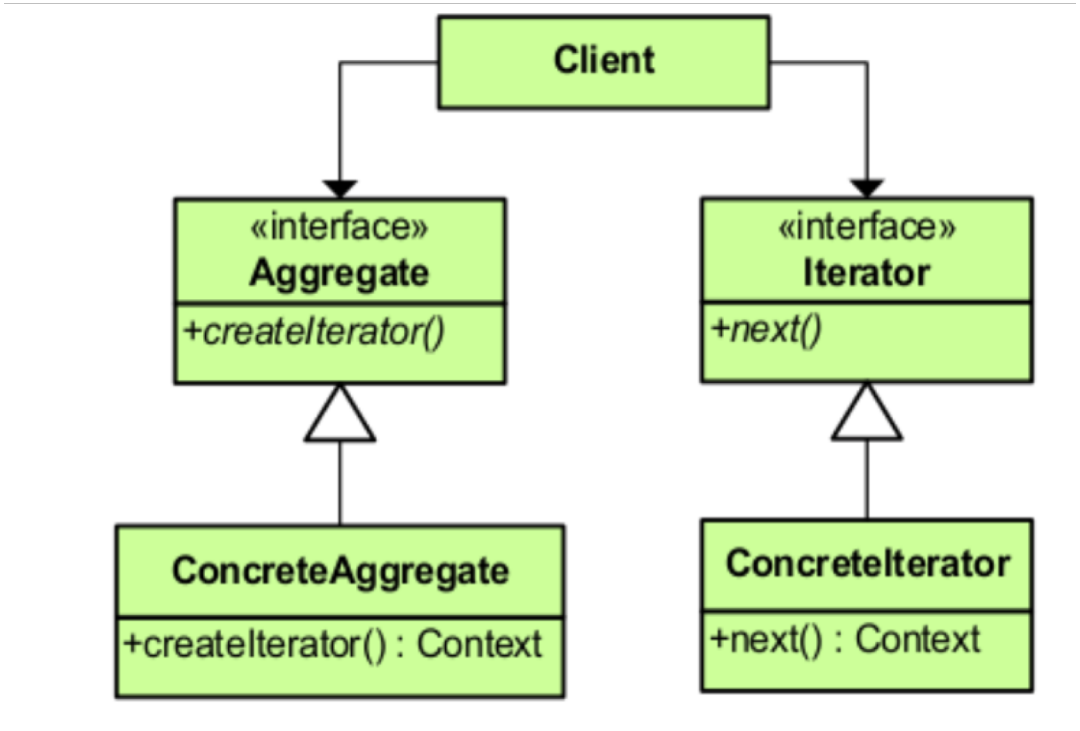
Quantifiers in Contracts

```
put(ar: ARRAY[REAL]; y: REAL; i: INTEGER)
    require
        ar.valid_index (i)
    do
        ar.put (y, i)
    ensure
        ar[i] = y
        across ar as  r all r.item <= 2 end
    end
```

$$\forall\ r \in ar:\ r \leq 2$$

- Will there be any contract violations?
- Who is to blame?

UML class diagram



Expanded vs. Reference

- Eiffel is strongly typed for readability and reliability. Every entity is declared of a certain type, which may be either a **reference** type or an **expanded** type.
- Any type *T* is based on a class, which defines the operations that will be applicable to instances of *T*. The difference between the two categories of type affects the semantics of using an instance of *T* as source of an *attachment* : assignment or argument passing.
- An attachment from an object of a reference type will attach a new reference to that object;
- with an expanded type, the attachment will *copy* the contents of the object

Expanded vs. Reference

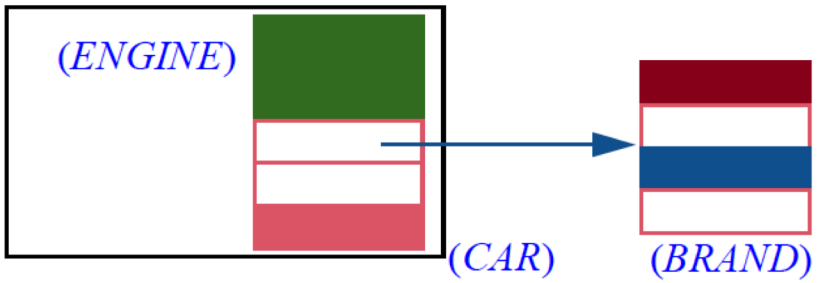
- With an expanded type, the attachment will *copy* the contents of the object.
- Similarly, comparison operations such as *a* = *b* will compare references in one case (references) and objects contents in the other.
- (To get object comparison in all cases, use *a* ~ *b*.)
- We talk of objects with *reference semantics* and objects with *value semantics* (or copy semantics).
- Syntactically, the difference is simple: a class declared without any particular marker, like *FOO*, yields a reference type. To obtain an expanded type, just start with **expanded** FOO

References as a modeling tool

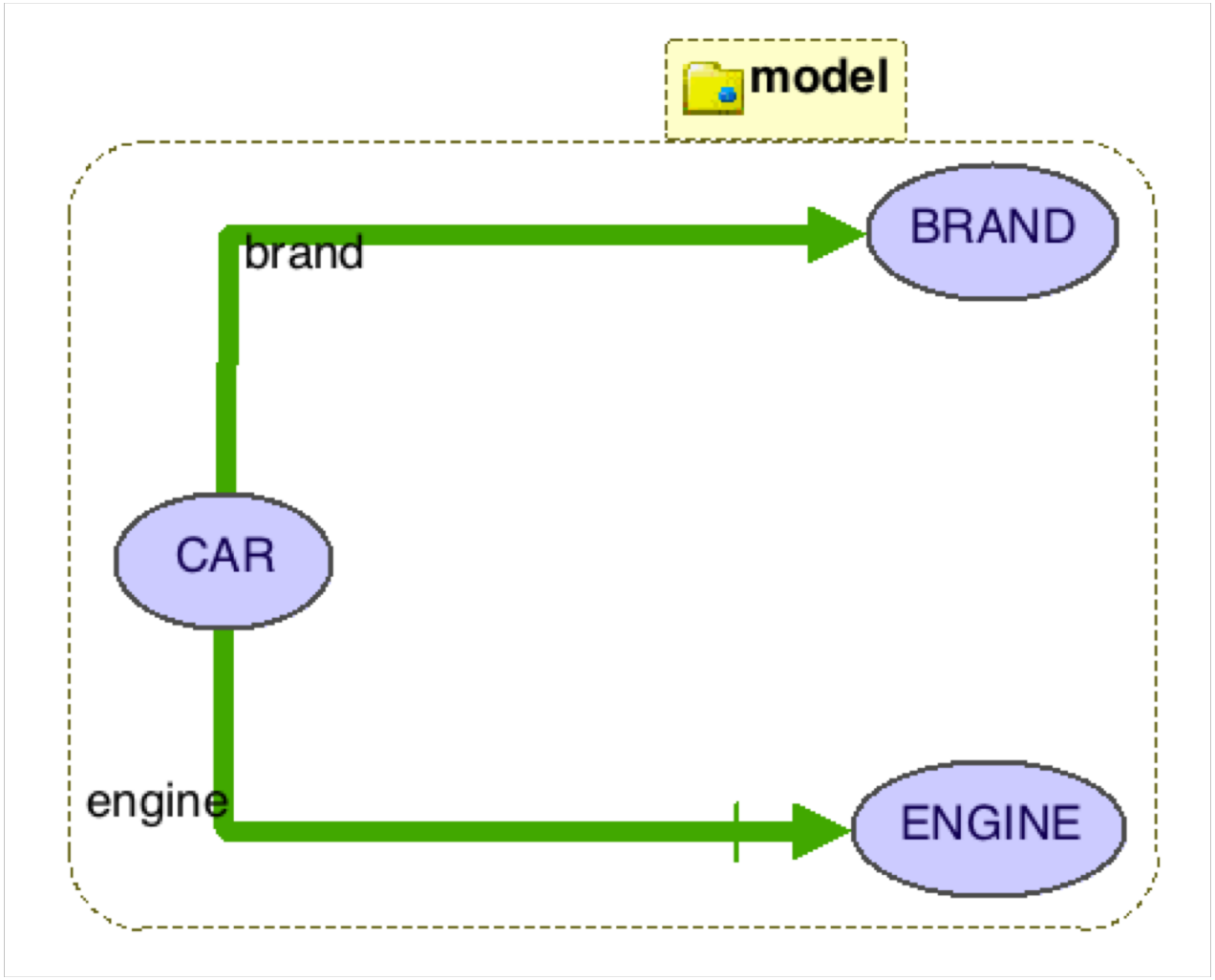
An object may include a reference to another object to represent the concept of "knowing about" that object, which you may compare with the concept of *containing* another object. Contrast for example two uses of the verb "to have" about cars:

- A car *has* an engine.
- A car *has* a brand.

The key difference is sharing: two cars may have the same brand (say they are both Nissans); but no self-respecting car would consent to sharing an engine with another. With object-oriented techniques we may model the first case through a subobject, the second through a reference to another object.



(*Expanded types* help model subobjects.) Such modeling flexibility is important in building programs that model complex systems.


model
BRAND
brand
CAR
engine
ENGINE

http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/expanded-car/

```
expanded class
        ENGINE

inherit
        ANY
                redefine
                        is_equal
                end

create
        default_create

feature

        horsepower: INTEGER_32

        drive: detachable STRING_8

        put (a_hp: INTEGER_32; a_drive: STRING_8)
                do
                        horsepower := a_hp
                        drive := a_drive
                end

        is_equal (other: like Current): BOOLEAN
                        -- Is `other' attached to an object considered
                        -- equal to current object?
                do
                        Result := horsepower = other.horsepower and drive ~ other.drive
                end

end -- class ENGINE
```

```
make
        -- (export status {NONE})
    local
        c1, c2: CAR
        b1, b2: BRAND
        e1, e2: ENGINE
    do
        create b1.make ("Bentley")
        b2 := b1
        e1.put (300, "AWD")
        e2 := e1
        create c1.make (2016, b1, e1)
        create c2.make (2016, b2, e2)
        check
                c1 /= c2 and b1 = b2 and e1 = e2
        end
        print ("%NNo contract violations")
    end
```

Feature | Class

Objects — {ROOT}.make

Name	Value	Type	Address
Current object	<0x10C06EF58>	ROOT	0x10C06EF58
Once routines			
Locals			
b1	Void	BRAND	Void
b2	Void	BRAND	Void
c1	Void	CAR	Void
c2	Void	CAR	Void
e1	<0x10C06EF40>	ENGINE	0x10C06EF40
drive	Void	NONE	Void
horsepower	0	INTEGER_32	
Once routi…			
e2	<0x10C06EF48>	ENGINE	0x10C06EF48
drive	Void	NONE	Void
horsepower	0	INTEGER_32	
Once routi…			

```
make
        -- (export status {NONE})
    local
        c1, c2: CAR
        b1, b2: BRAND
        e1, e2: ENGINE
    do
        create b1.make ("Bentley")
        b2 := b1
        e1.put (300, "AWD")
        e2 := e1
        create c1.make (2016, b1, e1)
        create c2.make (2016, b2, e2)
        check
            c1 /= c2 and b1 = b2 and e1 = e2
        end
        print ("%NNo contract violations")
    end
```

Feature | Class

Objects — {ROOT}.make

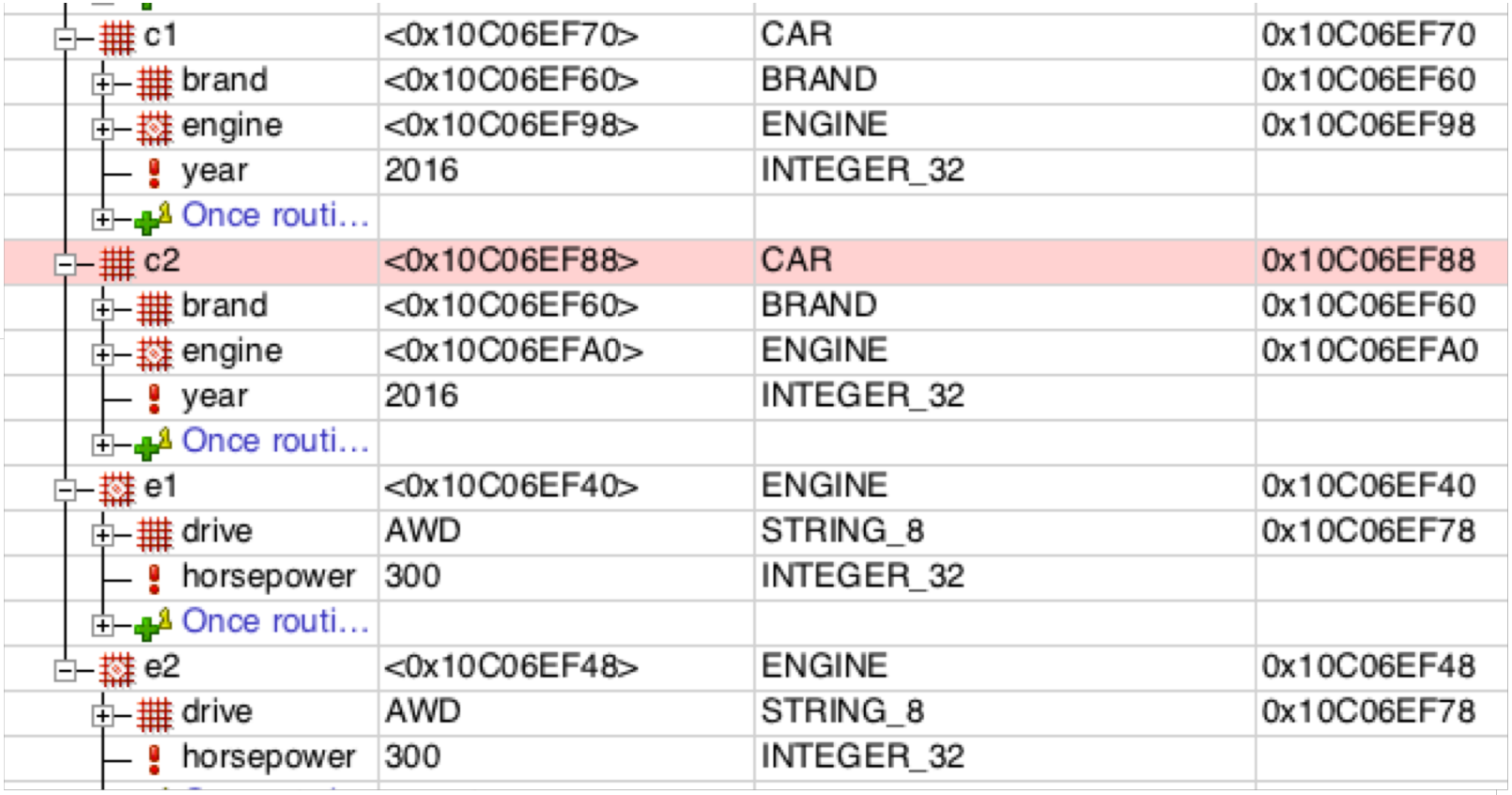
Name	Value	Type	Address
Current object	<0x10C06EF58>	ROOT	0x10C06EF58
Once routines			
Locals			
b1	<0x10C06EF60>	BRAND	0x10C06EF60
brand	Bentley	STRING_8	0x10C06EF68
Once routi…			
b2	<0x10C06EF60>	BRAND	0x10C06EF60
brand	Bentley	STRING_8	0x10C06EF68
Once routi…			
c1	Void	CAR	Void
c2	Void	CAR	Void
e1	<0x10C06EF40>	ENGINE	0x10C06EF40
drive	Void	NONE	Void
horsepower	0	INTEGER_32	
Once routi…			
e2	<0x10C06EF48>	ENGINE	0x10C06EF48
drive	Void	NONE	Void
horsepower	0	INTEGER_32	

```
make
        -- (export status {NONE})
    local
        c1, c2: CAR
        b1, b2: BRAND
        e1, e2: ENGINE
    do
        create b1.make ("Bentley")
        b2 := b1
        e1.put (300, "AWD")
        e2 := e1
        create c1.make (2016, b1, e1)
        create c2.make (2016, b2, e2)
        check
            c1 /= c2 and b1 = b2 and e1 = e2
        end
        print ("%NNo contract violations")
    end
```

Feature | Class

Objects — {ROOT}.make

Name	Value	Type	Address
Current object	<0x10C06EF58>	ROOT	0x10C06EF58
Once routines			
Locals			
b1	<0x10C06EF60>	BRAND	0x10C06EF60
brand	Bentley	STRING_8	0x10C06EF68
Once routi...			
b2	<0x10C06EF60>	BRAND	0x10C06EF60
brand	Bentley	STRING_8	0x10C06EF68
Once routi...			
c1	Void	CAR	Void
c2	Void	CAR	Void
e1	<0x10C06EF40>	ENGINE	0x10C06EF40
drive	AWD	STRING_8	0x10C06EF78
horsepower	300	INTEGER_32	
Once routi...			
e2	<0x10C06EF48>	ENGINE	0x10C06EF48
drive	AWD	STRING_8	0x10C06EF78
horsepower	300	INTEGER_32	



c1	<0x10C06EF70>	CAR	0x10C06EF70
brand	<0x10C06EF60>	BRAND	0x10C06EF60
engine	<0x10C06EF98>	ENGINE	0x10C06EF98
year	2016	INTEGER_32	
Once routi...			
c2	<0x10C06EF88>	CAR	0x10C06EF88
brand	<0x10C06EF60>	BRAND	0x10C06EF60
engine	<0x10C06EFA0>	ENGINE	0x10C06EFA0
year	2016	INTEGER_32	
Once routi...			
e1	<0x10C06EF40>	ENGINE	0x10C06EF40
drive	AWD	STRING_8	0x10C06EF78
horsepower	300	INTEGER_32	
Once routi...			
e2	<0x10C06EF48>	ENGINE	0x10C06EF48
drive	AWD	STRING_8	0x10C06EF78
horsepower	300	INTEGER_32	

```
        e1.put (300, "AWD")
        e2 := e1
        create c1.make (2016, b1, e1)
        create c2.make (2016, b2, e2)
        check
                c1 /= c2 and b1 = b2 and e1 = e2
        end
        print ("%NNo contract violations")
    end
```

Expanded

- An expanded class
  - uses the default_create constructor from class ANY (no arguments)
  - in an assignment e1 := e2, use copy semantics (for e1 and e2 instances of the expanded class ENGINE).
- Cars c1 and c2 share a BRAND but have distinct instances of ENGINE.
- Expanded: A car has-a (i.e. contains) an engine
- Reference: A car has-a (i.e. knows about) a brand (e.g. Acura, Bentley, etc.)

```
note
        description: "Arithmetic (+, -, *) with infinite precision"

expanded class VALUE inherit
        COMPARABLE
                redefine is_equal, default_create, out end
        NUMERIC
                redefine is_equal, default_create, out end
creation
        make_from_string,  default_create

feature {NONE} -- constructors
        default_create
                        -- create an empty object (equivalent to 0)
                local
                        empty: STRING
                do
                        empty := "0"; s := empty.twin
                ensure then s.is_equal ("0")
                end

        make_from_string (a_s: STRING)
                -- create a MONEY_VALUE object from string `s'
                require
                        non_empty: not s.is_empty
                        has_correct_format: ensureValid(s)
                do  s := a_s.twin; normalize end

feature
        plus alias "+", add (other: VALUE): VALUE
                        -- adds current to other and returns a new object

        minus alias "-", subtract (other: VALUE): VALUE
                        -- subtracts other from current and returns a new object

        product alias "*", multiply (other: VALUE): VALUE
                        -- multiplies current by other and returns a new object
        ...
end
```

Once objects
(seeTOC p687)

```
left_click: EVENT_TYPE [ TUPLE [x: INTEGER; y: INTEGER] ]
        -- Event type representing left-button click events
    once
        create Result
    end
```

- As the name suggests, a once routine (marked once instead of do or deferred) has its body executed at most once on its first call if any;
- subsequent calls will not execute any code and, in the case of a function, they will return the value computed by the first.
- One of the advantages is that you do not need to worry about when to create the object;
- whichever part of the execution first uses it will (unknowingly) do it.

```
expanded COMPLEX
```

```
i: COMPLEX
  once
     create Result.make(0,1)
  end
```

ONCE

Complex numbers: $-2 + \pi i$, where i is the imaginary unit

```
i: COMPLEX
  do
    create Result.make(0,1)
  end
```

The function routine *i* will always return a reference to an object representing the imaginary unit. However, each client produces a new object identical to all the others.

```
i: COMPLEX
  once
    create Result.make(0,1)
  end
```

The first time a once function is called during a system execution, it executes the body. Every subsequent execution, executes no instructions at all; it just returns the result computed first time round.

Singleton Pattern

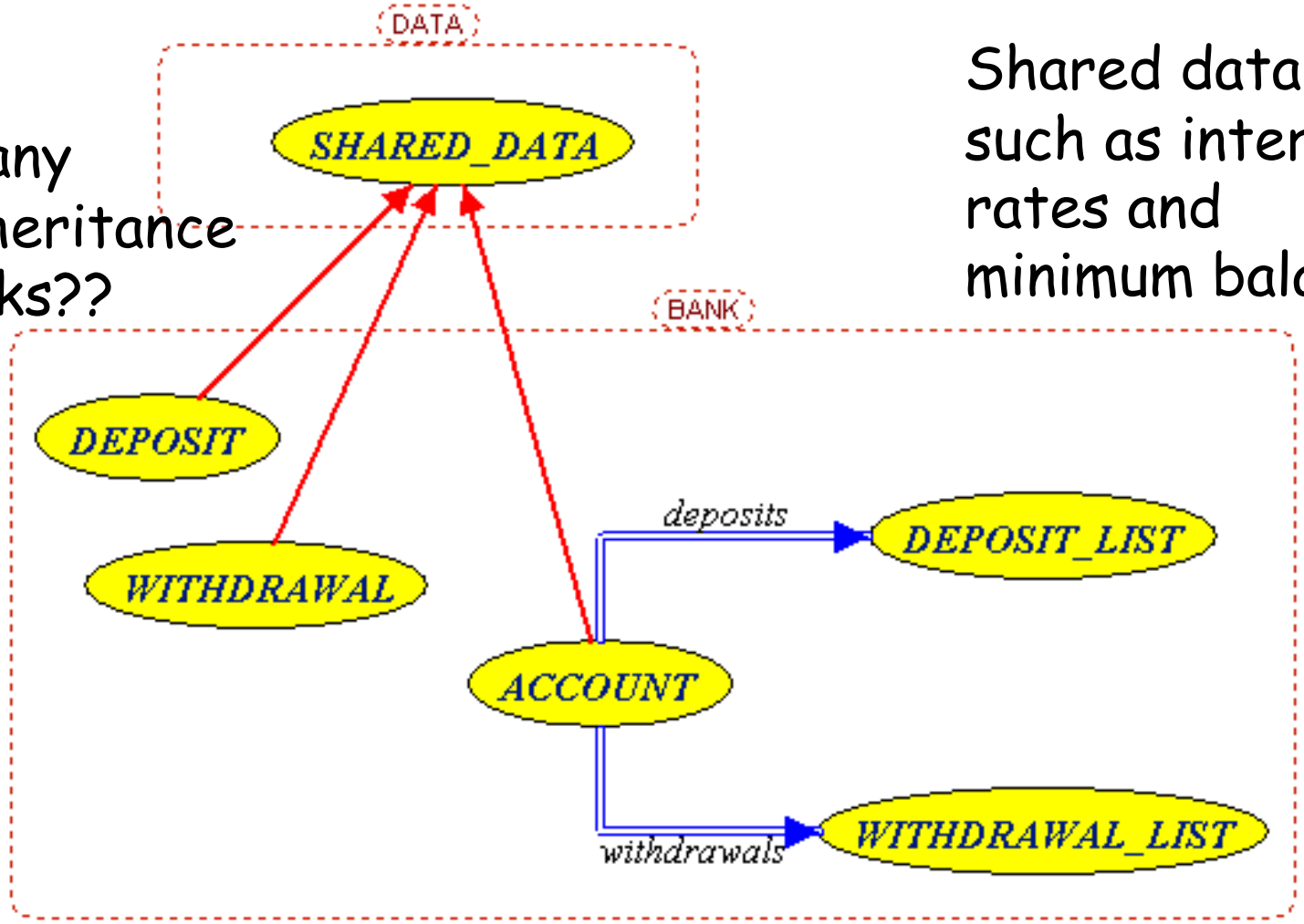
- Sometimes it's appropriate to have exactly one instance of a class:
  - window managers,
  - print spoolers,
  - and filesystems
- Typically, those types of objects—known as singletons—are accessed by disparate objects throughout a software system, and therefore require a global point of access.

Singleton Pattern

- With the Singleton design pattern you can:
  - Ensure that only one instance of a class is created
  - Provide a global point of access to the object
  - Allow multiple instances in the future without affecting a singleton class's clients

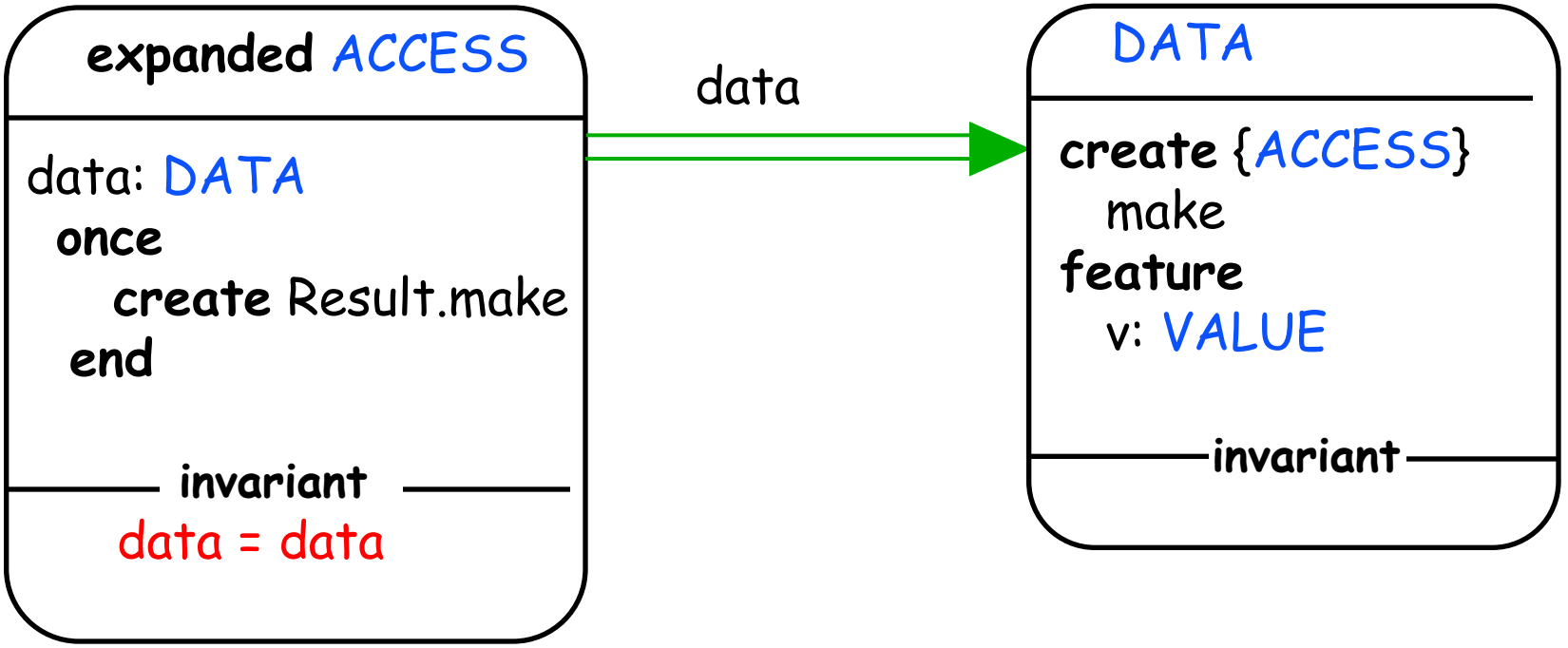
Need for shared data

Many inheritance links??



Shared data such as interests rates and minimum balance

The Singleton Pattern



a: ACCESS; value: VALUE
value := a.data.v

OR

value := **create** {ACCESS}.data.v
-- can be done as many times as you like

create {DATA}.make ;
-- will not compile

Java Singleton

```
public class ClassicSingleton {
   private static ClassicSingleton instance = null;
   protected ClassicSingleton() {
      // Exists only to defeat instantiation.
   }
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

The static keyword in Java means that the variable or function is shared between all instances of that class as it belongs to the type, not the actual objects themselves. So if you have a variable: private static int i = 0; and you increment it (i++) in one instance, the change will be reflected in all instances