Marktoberdorf 2004

Towards Trusted Components

Bertrand Meyer

ETH, Zürich & Eiffel Software, California

Lesson 3:
Strategy for proving classes

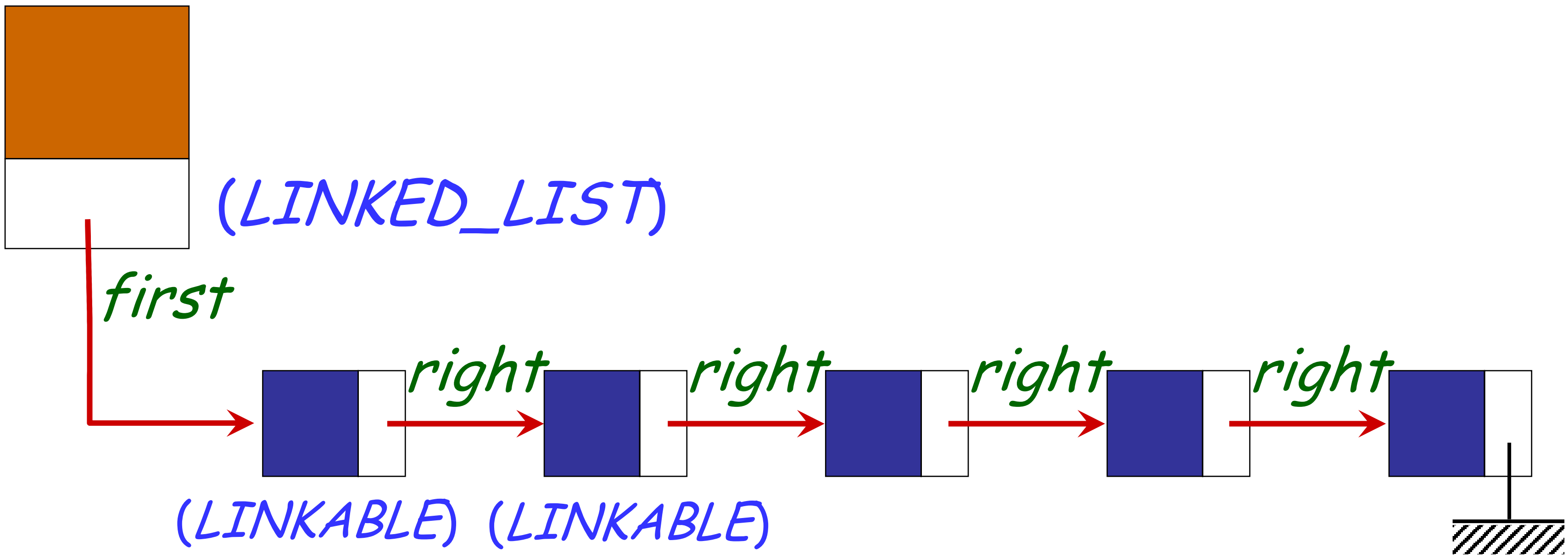
What we do with contracts today

- Specify, design, implement
- Document
- Test & debug
- Control inheritance, exceptions
- Manage

The next step

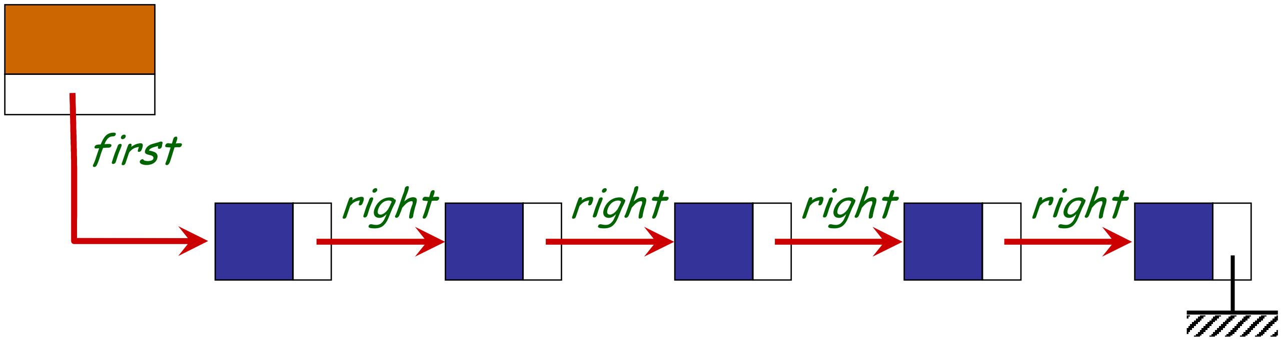
Prove that class implementations satisfy the contracts

Linked lists in EiffelBase



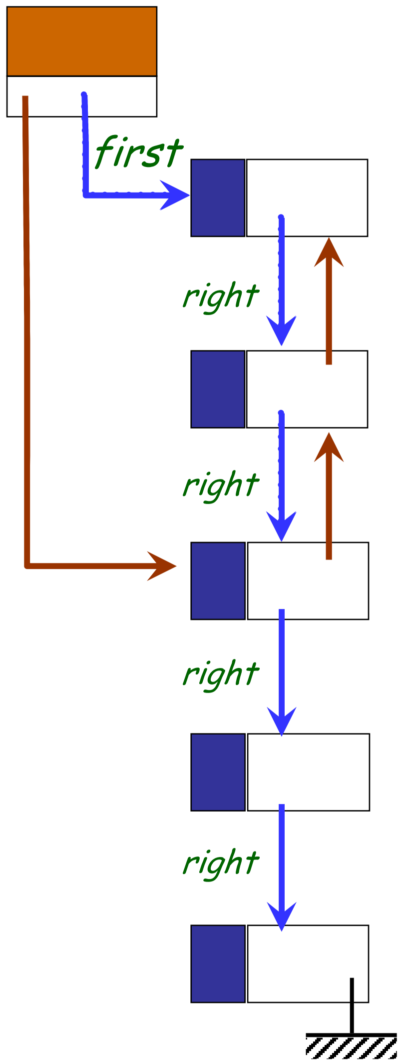
Some of what we must prove

- Starting from *first* and following *right* links:
  - No element encountered twice
  - Eventually reaches a *Void*
- An insertion keeps the previous elements:
  - Left of insertion, with same index as before
  - Right of insertion, with previous index plus 1



Reversing a list

```
reverse is
    local
        previous, next: LINKABLE [G]
    do
        from
            next := first ; first := Void
        invariant
            ...
        until next = Void loop
            previous, first, next := [first, next, next.right]
            first.put_right (previous)
        end
    ensure
        ...
    end
```



The framework

Like engineers of traditional fields:

- We are building a system
- We want to guarantee its precise properties
- We devise a model and prove it has these properties

Unlike them:

- We define and completely control the product:

The system *is* the model!

(except for dependencies on hardware and other software)

Principles

Very simple mathematics only, few "rabbits"

More principles

Work on mathematical representation, not program text

(Avoid "symbol pushing")

Mix of

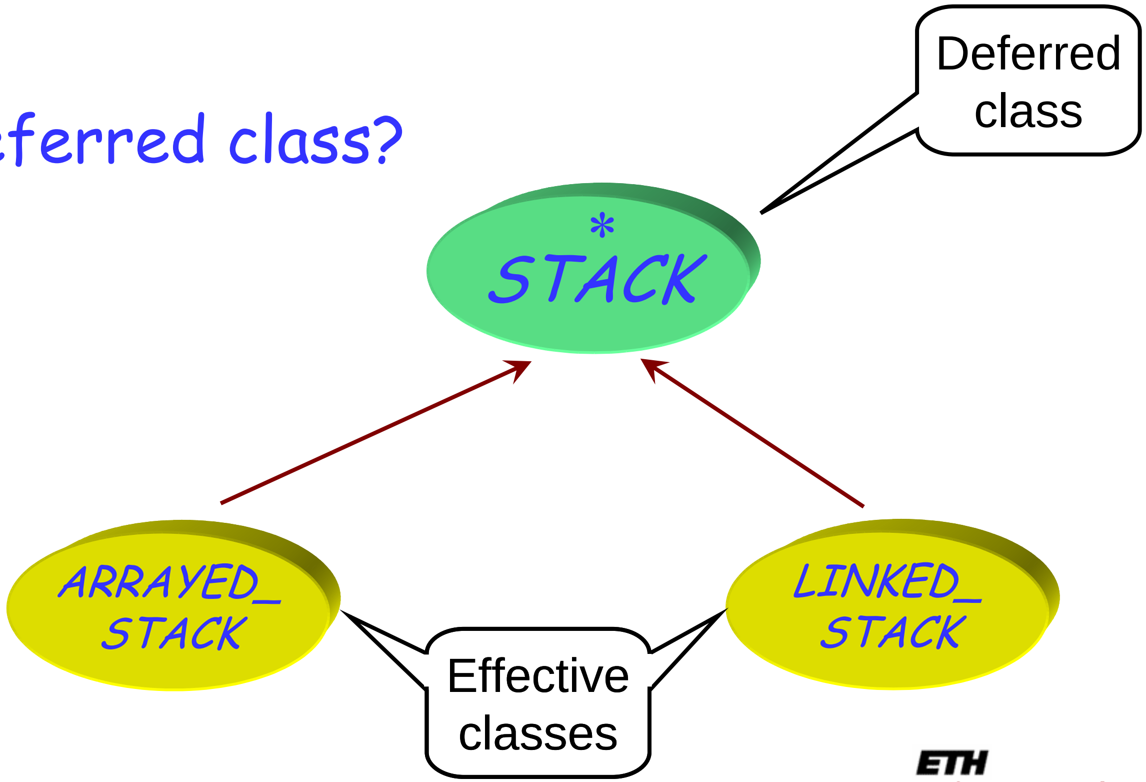
- Denotational
- Axiomatic

"Proving a class"

Not an abuse of language in Eiffel because classes contain their own contracts

How to deal with inheritance: friend or foe?

Can we "prove" a deferred class?



A deferred class with contracts

```
deferred class
    STACK [G]

feature -- Access

    count : INTEGER
            -- Number of stack items

    item : G is
            -- Top element
        require
            count > 0
        deferred
        end

    empty : BOOLEAN
            -- Are there no items?

    full : BOOLEAN
            -- Is there no more room?

feature -- Element change
    put (x : G) is
            -- Push x to top of stack.
        require
            not full
        deferred
        ensure
            item = x
            count = old count + 1

        end
    remove is
            -- Pop top of stack.
        require
            not empty
        deferred
        ensure
            count = old count – 1

        end
end
```

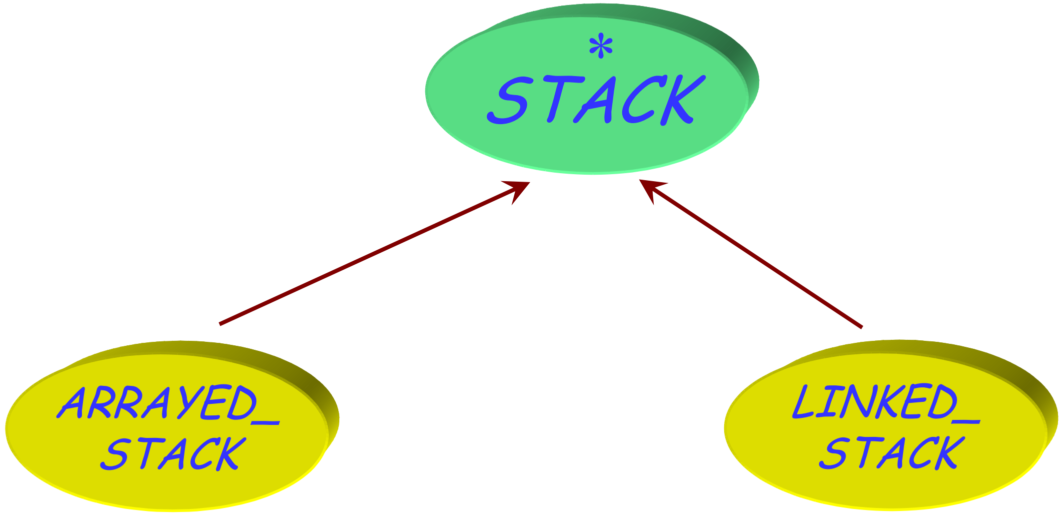
An implementation (effective class)

```
deferred class
    BOUNDED_STACK [G]

inherit

    STACK [G]

    ARRAY [G]

...

feature -- Element change
    put (x: G) is
            -- Push x to top of stack.

        do
            count := count + 1
            item [count] := x

        end

    remove is
            -- Pop top of stack.

    do
        count := count — 1

    end
```

Stating the problem

What does it mean to "prove"

- The deferred class?
- The effective class?



Providing a full specification

Contract language: Boolean expressions of Eiffel, plus **old** keyword in postconditions

The postconditions of contracts in EiffelBase are often not complete

In *STACK* [*G*] as shown earlier:

```
put (x : G) is
        -- Push x to top of stack.
    require
        not full
    deferred
    ensure
        item = x
        count = old count + 1
    end
```

We do *not*, however, expand the power of the contract language!

Using models

Eiffel Model library (see similar approach in JML):

Classes

SET

RELATION

FUNCTION,

TOTAL_FUNCTION

SEQUENCE…

Totally applicative: functions only, no side effects, no assignments

Example:

SEQUENCE [*G*] denotes finite sequences of items of type *G*

(Formally: functions from 1..*n* to *G* for some integer *n*)

Some features in class *SEQUENCE [G]*

All are queries:

- *tail* : *SEQUENCE [G]*

  -- Same items except first

  require

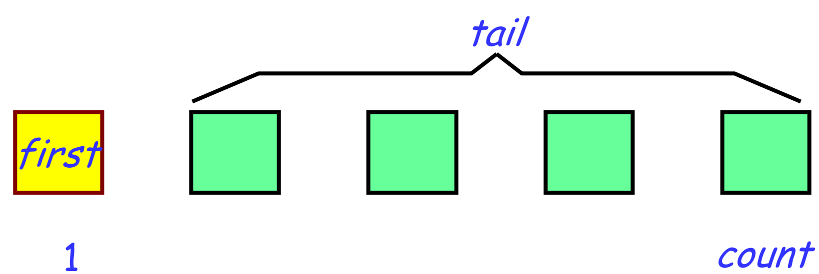
  not empty

- *first* : *G*

  -- First element

  require

  not empty



- *prepended* (*x*: *G*): *SEQUENCE [G]*

  -- Same items, plus *x* added at beginning

  ensure

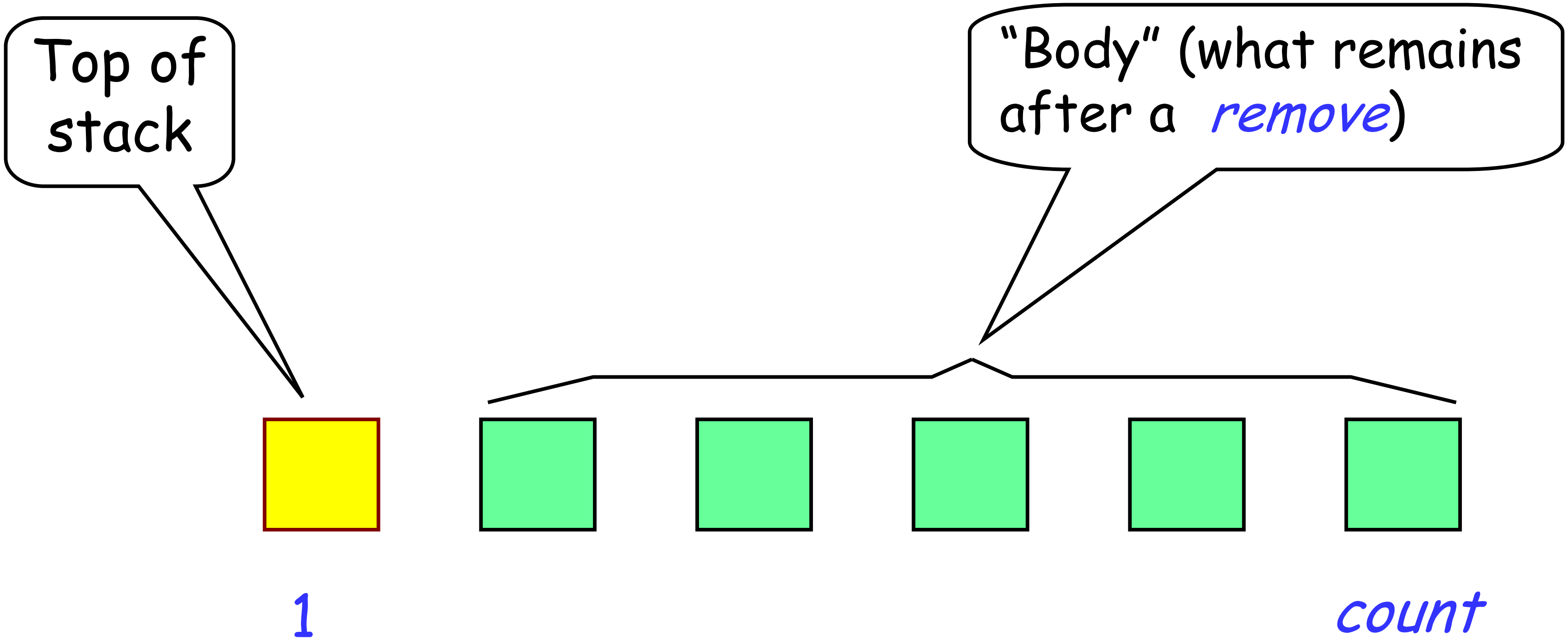
  Result.first ~ *x*

  Result.tail ~ *Current*

Object equality

Mathematically modeling a software notion

Example: model a stack as a sequence



A deferred class with contracts

```
deferred class
    STACK [G]
feature -- Specification
    model: SEQUENCE [G]
feature -- Access

    count: INTEGER
            -- Number of stack items

    item: G is
            -- Top element
        require
            count > 0
        deferred
        end

    empty: BOOLEAN
            -- Are there no items?

    full: BOOLEAN
            -- Is there no more room?
```

Model contract

```
feature -- Element change
    put (x: G) is
            -- Push x to top of stack.
        require
            not full
        deferred
        ensure
            item = x
            count = old count + 1
            model = old model.prepended (x)
        end
    remove is
            -- Pop top of stack.
        require
            not empty
        deferred
        ensure
            count = old count – 1
            model = old model.tail
        end
end
```

Abstract contract

Proofs (1)

At the deferred class level:

- Prove that the model contracts imply the abstract contracts

An implementation

```
deferred class
    BOUNDED_STACK [G]

inherit

    STACK [G]

    ARRAY [G]

...
```

```
feature -- Element change
    put (x: G) is
            -- Push x to top of stack.

        do
            count := count + 1
            item [count] := x

        end


    remove is
            -- Pop top of stack.


        do
            count := count — 1

        end
```

Proofs (2)

At the deferred class level:

- Prove that the model contracts imply the abstract contracts

At the effective class level:

- Prove that the implementation satisfies the model contracts

The Current Calculus

Set of operators to deal with the special nature of object-oriented programming

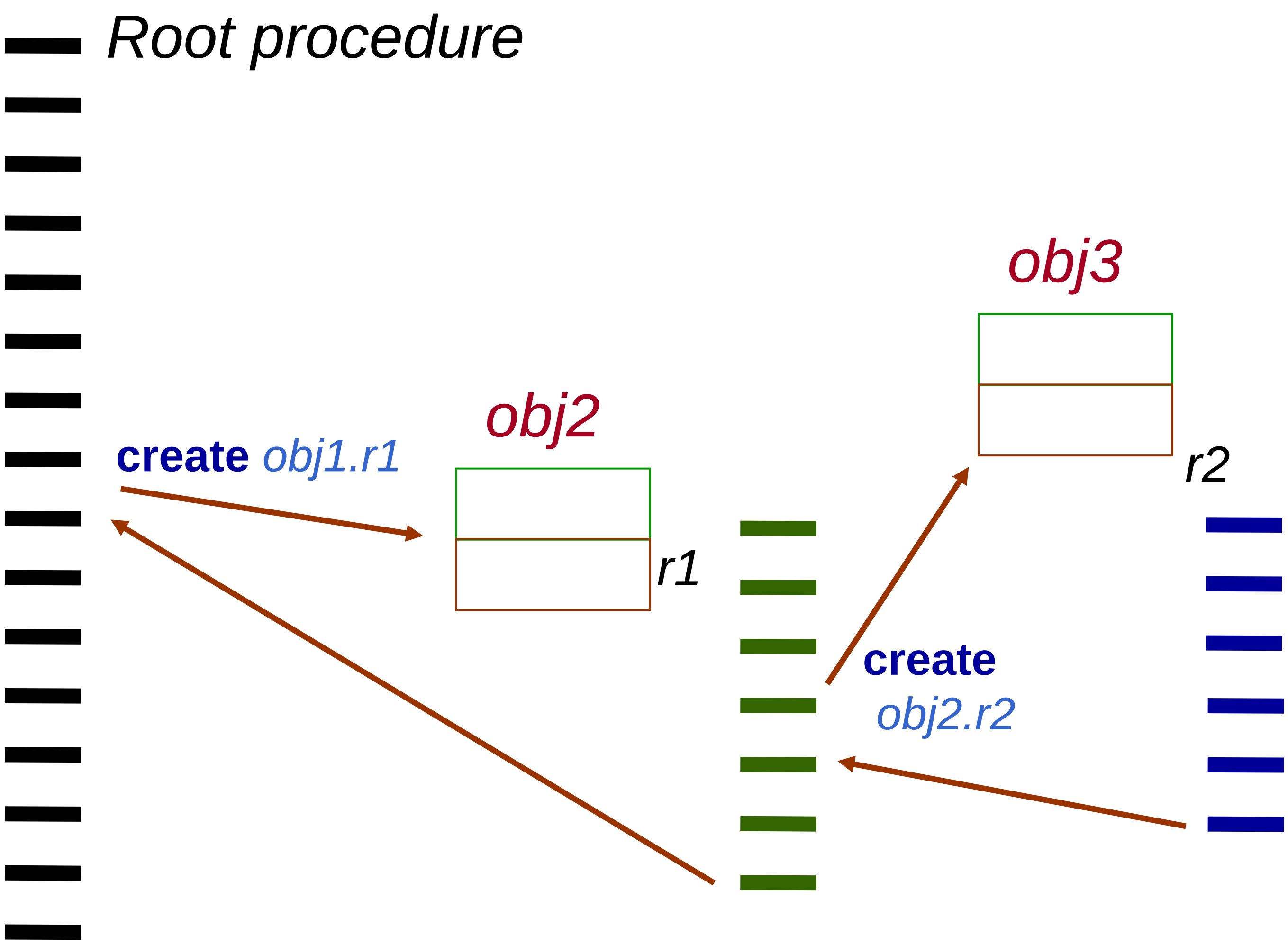
Basic operation:

$$x.f(a)$$

with $x : C$ for some class C

"**Principle of general relativity**": everything you write refers to the current object

Executing an Eiffel system



The Current Calculus (CC)

Classical denotational specifications:

$$some\_function : States \rightarrow Some\_set$$

In CC:

$$some\_oo\_function : States \rightarrow Objects \rightarrow Some\_set$$

Summary of lesson 3

"Proving a class" means proving that it satisfies its contracts

A simple theoretical framework seems sufficient: sets, relations, functions (total, possibly partial).

To make proofs convincing we should avoid special notations

We can obtain complete specifications through models

We can express everything — specification and implementation — in a single framework (here Eiffel)

The special nature of O-O programs ("general relativity") requires appropriate mathematical operators