Strong specifications for API design

Nadia Polikarpova

Software Architecture, Spring 2011

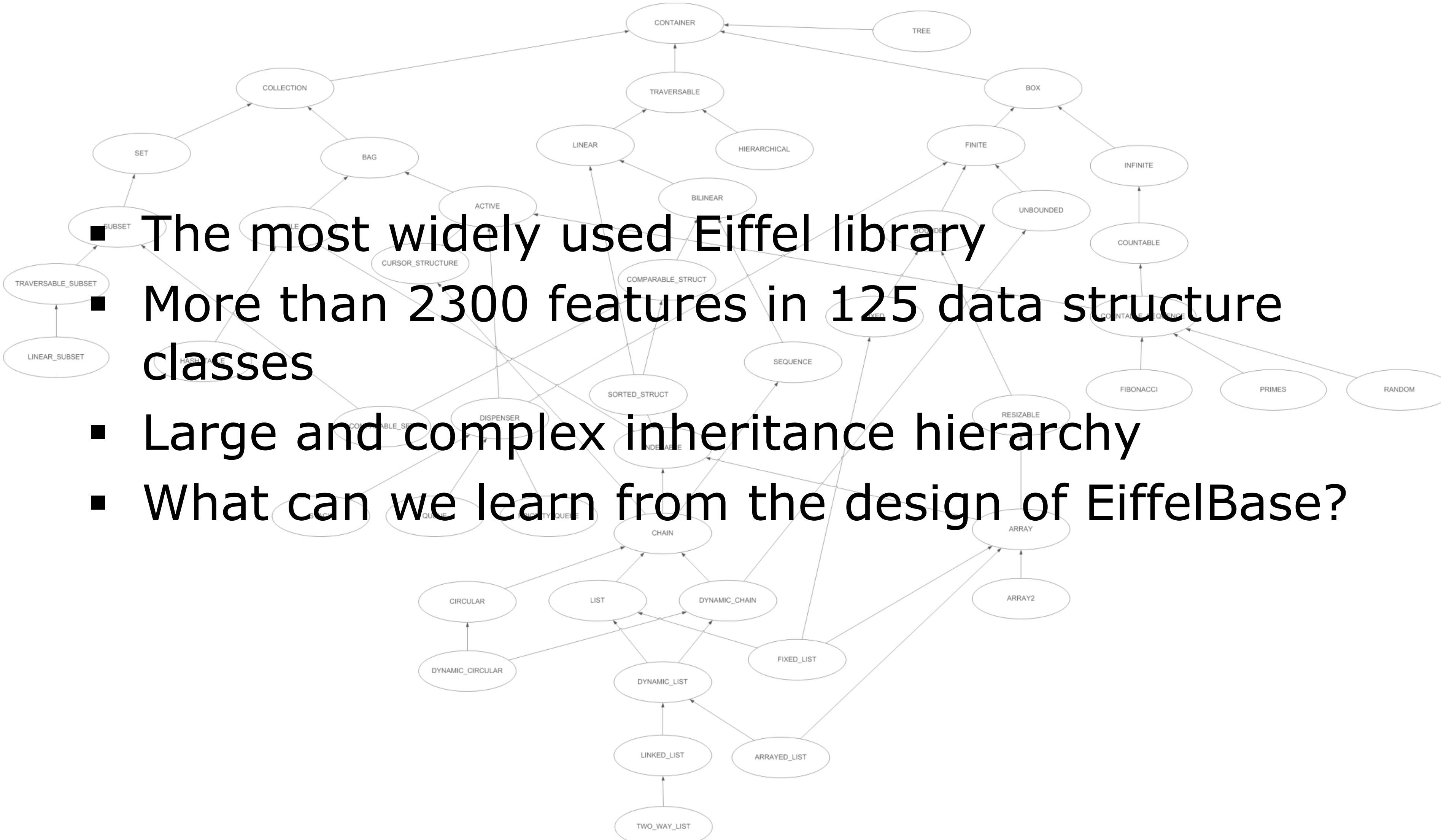
API design principles*

* Joshua Bloch, “How to Design a Good API and Why it Matters”

- API should do one thing and do it well
- Implementation should not impact API
- Document state space very carefully
- Document contract between method and its client
- Subclass only where it makes sense
- User of API should not be surprised by behavior
- Report errors as soon as possible after they occur

- Informal principles
- Easy to state, (sometimes) hard to follow
- Can strong formal specifications help?

EiffelBase

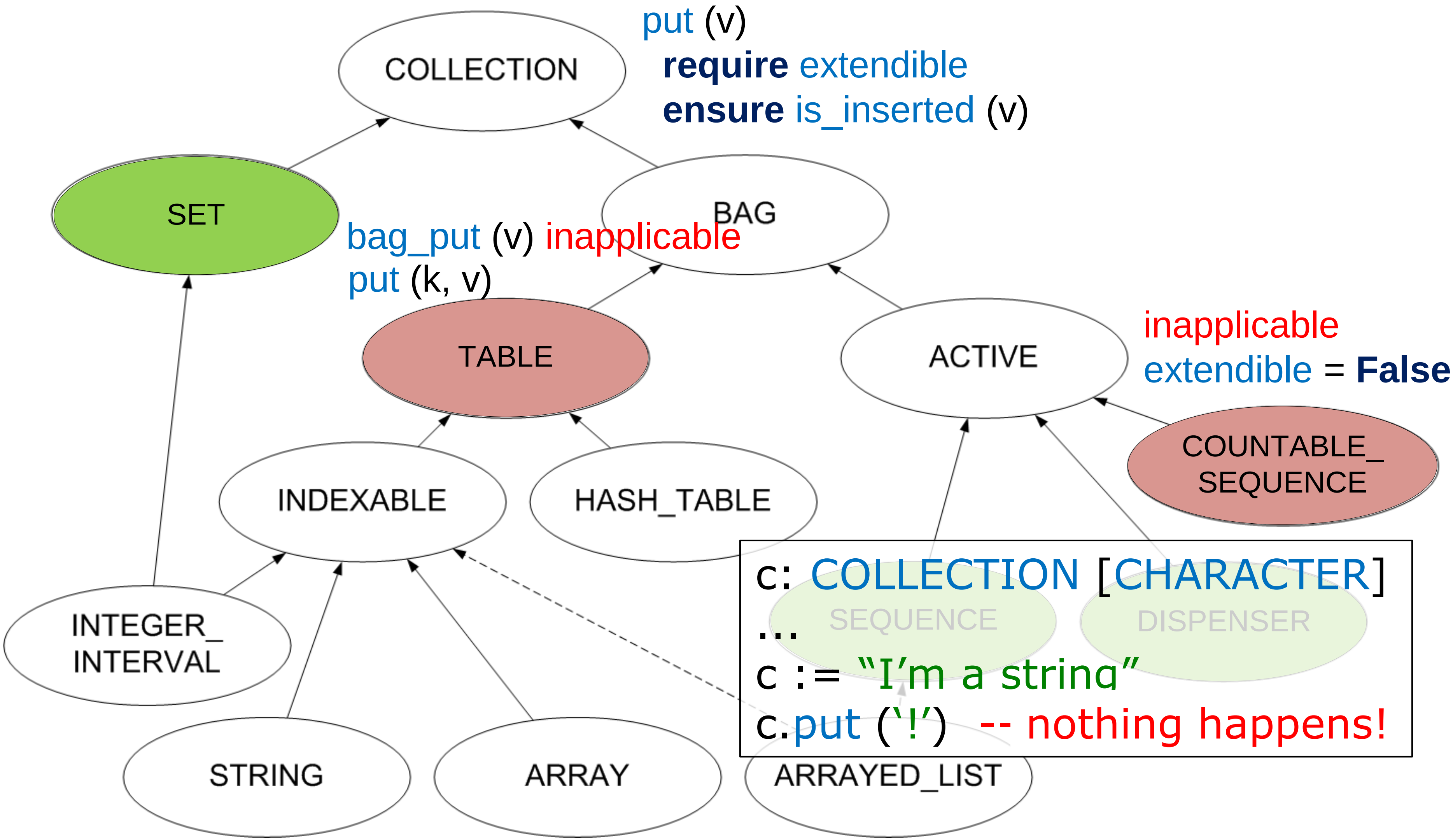


- The most widely used Eiffel library
- More than 2300 features in 125 data structure classes
- Large and complex inheritance hierarchy
- What can we learn from the design of EiffelBase?

Example: COLLECTION


```
deferred class COLLECTION [G] inherit CONTAINER [G]

	extendible: BOOLEAN -- May new items be added?

	prunable: BOOLEAN -- May items be removed?

	is_inserted (v: G): BOOLEAN
		-- Has `v' been inserted by the most recent insertion?

	put (v: G) -- Ensure that structure includes `v'.
		require extendible
		ensure is_inserted (v)

	prune (v: G) -- Remove one occurrence of `v' if any.
		require prunable

	wipe_out -- Remove all items.
		require prunable
		ensure is_empty
end
```

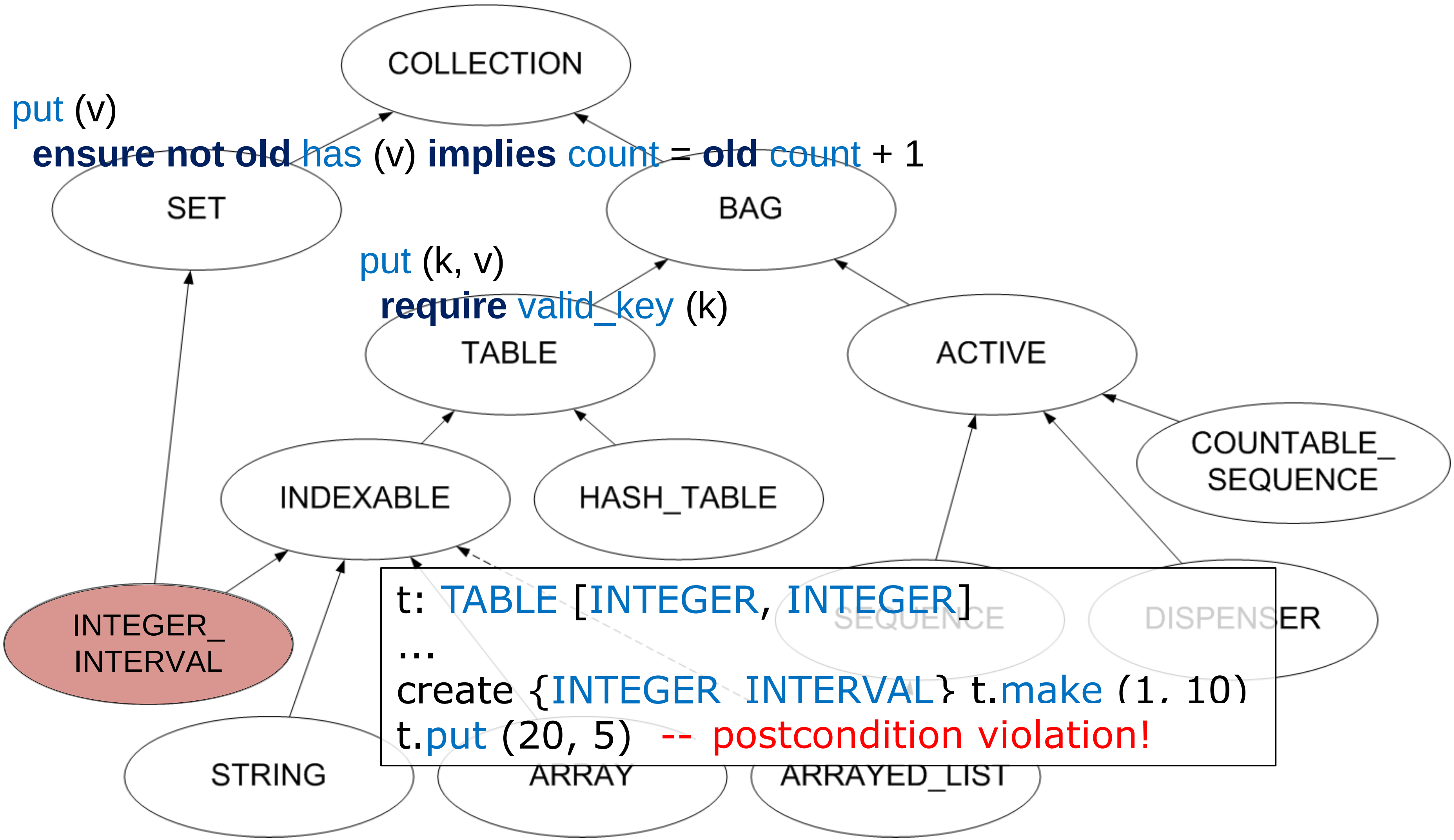
The *put* with a thousand faces (1)



The *put* with a thousand faces (2)

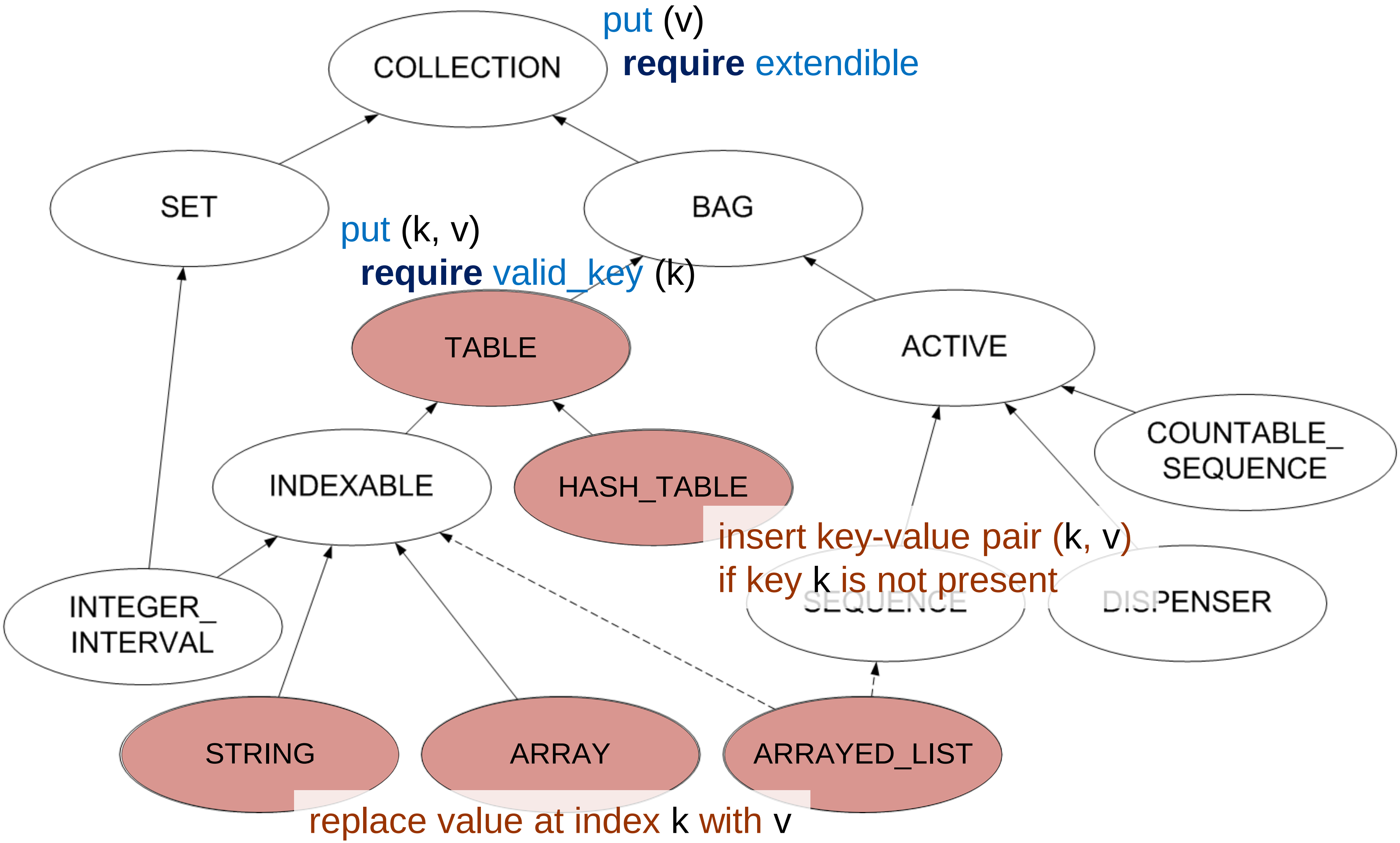
put (v)
ensure not old has (v) **implies** count = **old** count + 1

put (k, v)
require valid_key (k)



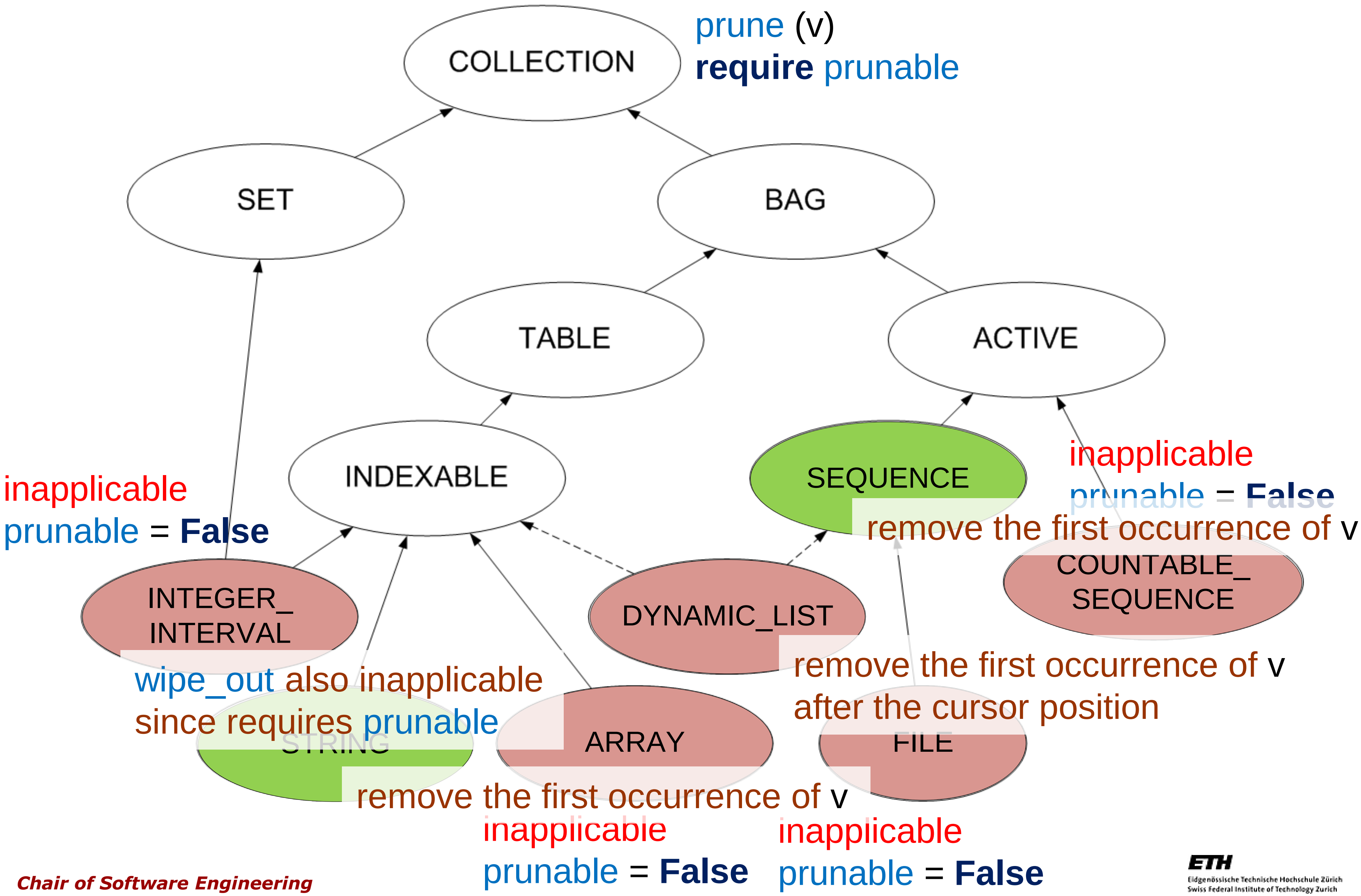

```
t: TABLE [INTEGER, INTEGER]
...
create {INTEGER_INTERVAL} t.make (1, 10)
t.put (20, 5)  -- postcondition violation!
```

The *put* with a thousand faces (3)



The *prune* with a thousand faces

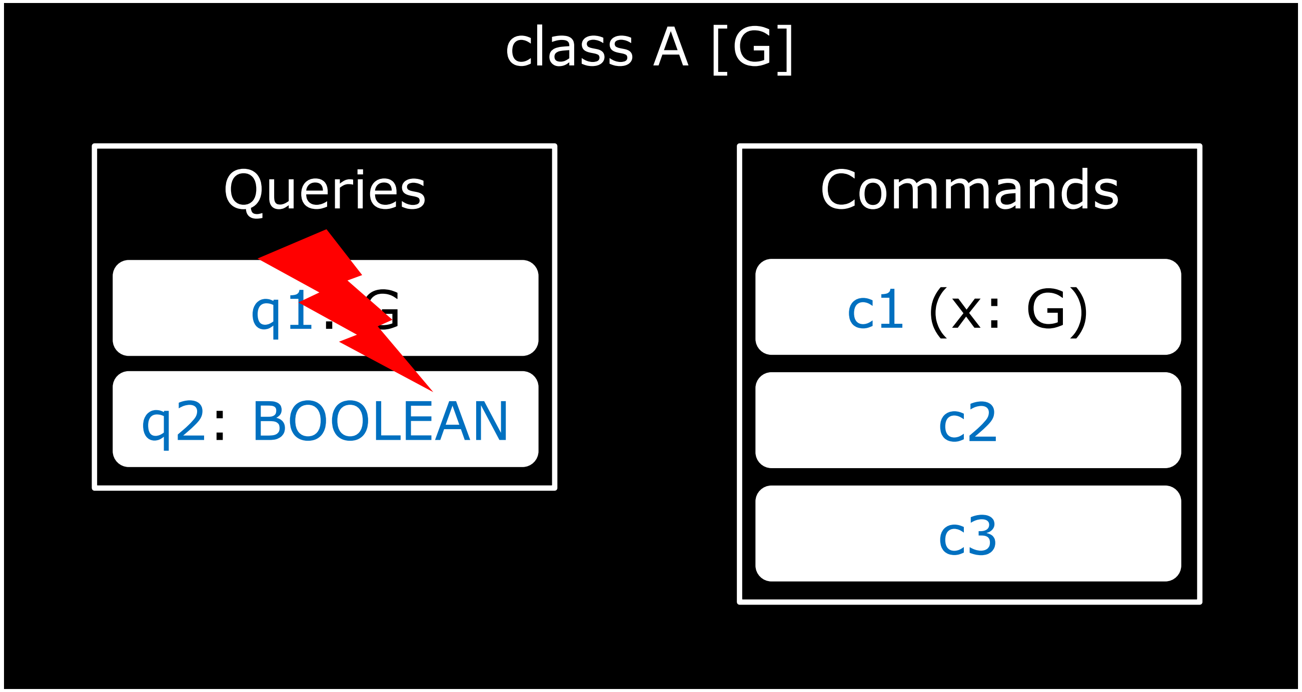
prune (v)
require prunable



EiffelBase: observations

- Deferred classes have vague semantics
  - about 1/3 features in class LIST have no postcondition or related invariant clause
  - often "placeholders" like extendible and prunable
- Many features of ancestors are inapplicable in descendants
  - 31 features in EiffelBase.structures are explicitly marked "Inapplicable"
  - even more with precondition **False**
- The semantics is often inconsistent among descendants

API with no contracts




```
a: A [INTEGER]
...
print (a.q1)
```

API with vague contracts

class A [G]

Queries	Commands
q1: G **require** q1_able	c1 (x: G)
q2: BOOLEAN	c2 **require** c2_able
q1_able: BOOLEAN	c3
c2_able: BOOLEAN	

```
a: A [INTEGER]
...
if a.q1_able then
    print (a.q1)
else
    -- ?
end
```

API with model-based contracts

class A [G]

Abstract state

s: SEQUENCE [G]

Queries

q1: G
require not s.is_empty
ensure Result = s.last

q2: BOOLEAN
ensure Result = s.is_empty

Commands

c1 (x: G)
ensure
s = **old** s.extended (x)

c2
require
not s.is_empty
ensure
s = **old** s.but_last

c3
ensure s.is_empty

API with model-based contracts

class STACK [G]

Abstract state

s: SEQUENCE [G]

Queries

top: G
require not s.is_empty
ensure Result = s.last

is_empty: BOOLEAN
ensure Result = s.is_empty

Commands

push (x: G)
ensure
s = **old** s.extended (x)

pop
require
not s.is_empty
ensure
s = **old** s.but_last

wipe_out
ensure s.is_empty

Model-based contracts (MBC)

```
note model: sequence
class STACK[G]
   sequence: MML_SEQUENCE [G]
         -- Sequence of elements.
   is_empty: BOOLEAN
         -- Is the stack empty?
      ensure Result = sequence.is_empty
   top: G
         -- Top of the stack.
      require not sequence.is_empty
      ensure Result = sequence.last
   push (v: G)
         -- Push `v' on the stack.
      ensure sequence = old sequence.extended (v)
   pop
         -- Pop the rop of the stack.
      require not sequence.is_empty
      ensure sequence = old sequence.but_first
```

complete

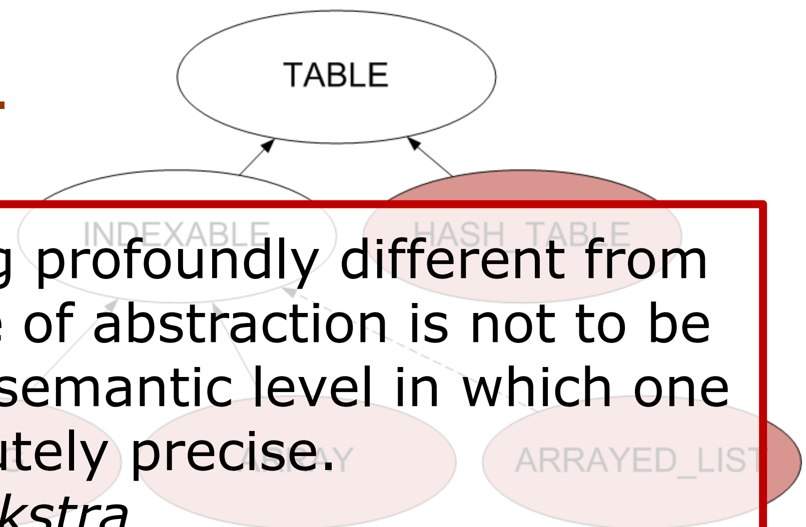
complete

MBC for reusable components

- Models make the abstract state space of the class explicit
  - give clients and developers intuition “how to think” about the class
  - using standard mathematical objects as models improves understanding
- Completeness can be defined and analyzed for model-based contracts
  - violation of completeness are a hint for the developer
  - complete contracts prevent inconsistencies in inheritance hierarchies

Example: TABLE.put

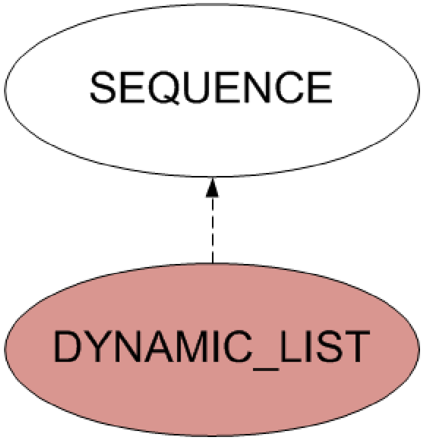
```
note
    model: map
deferred class TABLE [K, V]
    put (k: K; v: V)
            -- Associate value `v' with key `k'.
        require map.domain.has (k)
        ensure map = old map.replaced (k, v)
    ...
    map: MML_MAP [K, V]
            -- Map of keys to values.
end
```



> Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> *E. Dijkstra*

Example: SEQUENCE.prune

```
note
    model: sequence, index
deferred class SEQUENCE [G]
    prune (v: G)
            -- Remove the first occurrence of `v'.
        ensure
            sequence = old (sequence.removed_at
                (sequence.index_of (v)))
            index = old (sequence.index_of (v))
    ...
    sequence: MML_SEQUENCE [G]
            -- Sequence of elements.
end
```



Testing experiment

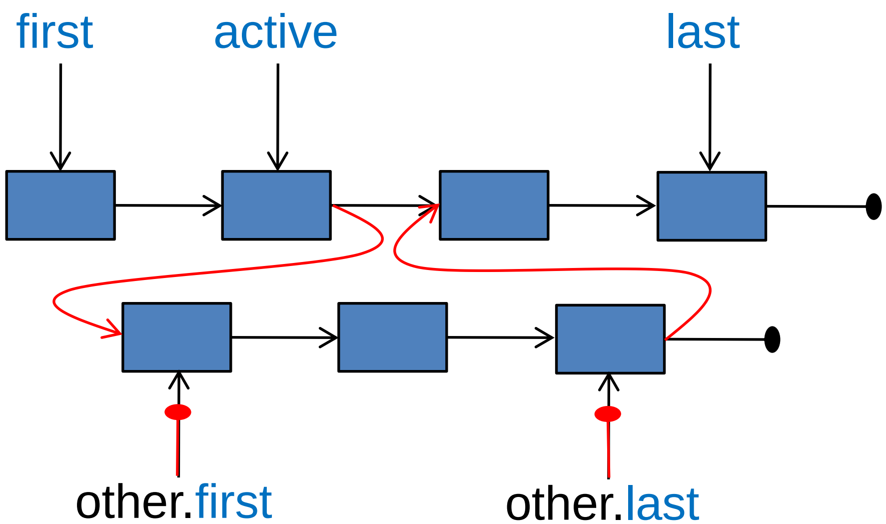
- Added MBC to a subset of EiffelBase

  7 flattened classes, 254 public methods, 5750 LOC
- Debugging revealed 3 faults in the implementation
- Automatic random testing against MBC for 30 minutes revealed 1 more fault (shown next)
- All 4 failing test cases would not violate original contracts

A larger class of faults is testable against complete model-based contracts

Fault example (1)

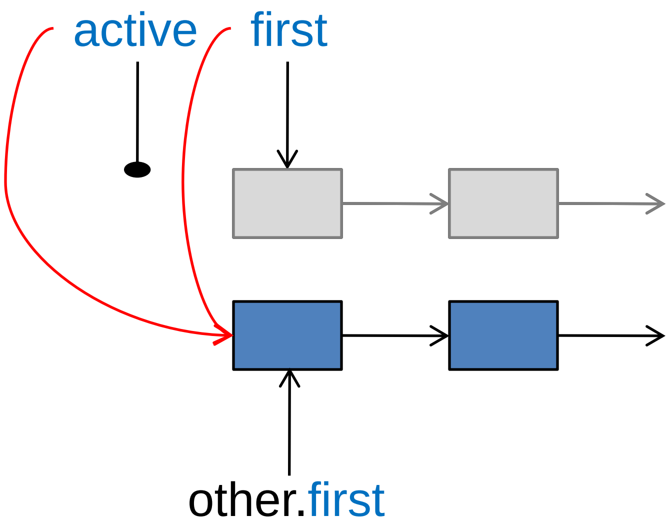

```
merge_right (other: LINKED_LIST [G])
    -- Merge `other' into current structure after cursor
    -- position. Do not move cursor. Empty `other'.
  require not after
```




```
  ensure
    new_count: count = old count + old other.count
    same_index: index = old index
    other_is_empty: other.is_empty
    sequence_effect: sequence = old (sequence.front (index) +
          other.sequence + sequence.tail (index + 1)))
  end
```

Fault example (2)


```
merge_right (other: LINKED_LIST [G])
    -- Merge `other' into current structure after cursor
    -- position. Do not move cursor. Empty `other'.
  require not after
  do ...
    if active = Void then
      first := other.first
      active := first
    else ... end
    count := count + other.count
    ...
  ensure
    new_count: count = old count + old other.count
    same_index: index = old index
    other_is_empty: other.is_empty
    sequence_effect: sequence = old (sequence.front (index) +
          other.sequence + sequence.tail (index + 1)))
  end
```



EiffelBase2: goals and results

- Verifiability
- ✓ Simple and consistent hierarchy: avoid "overabstraction" and "taxomania"
- ✓ Complete model-based contracts

	EiffelBase2	EiffelBase
Classes	57	125
Features	537	2300
hidden by descendants	0	31
with incomplete contract	5%	(LIST) 65%
with no contract	0%	(LIST) 30%

EiffelBase2: try it!

http://eiffelbase2.origo.ethz.ch

Conclusions

- Reusable components need strong specifications even on high levels of abstraction
- Model-based contracts is an effective approach to writing strong specifications in Eiffel
- Definition of completeness can be used to reason whether model-based contracts are strong enough
- Complete contracts prevent behavioral inconsistencies in class hierarchies
- EiffelBase2 case study has shown that writing strong model-based contracts is feasible
- Testing against stronger contracts reveals more faults

API design principles revisited

API should do one thing and do it well

All features operate on a single model

Implementation should not impact API

Semantics of all features in terms of abstract state

Document state space very carefully

Model documents abstract state space formally

Document contract between method and its client

Complete pre- and postconditions

Subclass only where it makes sense

Complete contracts prevent from subclassing when feature semantics is inconsistent

User of API should not be surprised by behavior

User relies on complete contracts

Report errors as soon as possible after they occur

Strong contracts reveal even subtle faults in a localized way