Principles of Library Design: The Eiffel Experience

Bertrand Meyer

ADFOCS Summer School, 2003

LECTURE 1

1

"Plan"

- 1: Intro to Eiffel and Principles of library design
- 2: Design by Contract
- 3: Trusted Components
- 4: Towards proofs

2

My background

- Since 2001: Professor of Software Engineering at ETH Zürich
- Since 1985: Founder (now Chief Architect) of Eiffel Software, in Santa Barbara. Produces Eiffel tools and services
- Also adjunct professor at Monash University in Australia (since 1998)

3

Lesson 1: Principles of Library Design

- Building effective libraries of reusable components

4

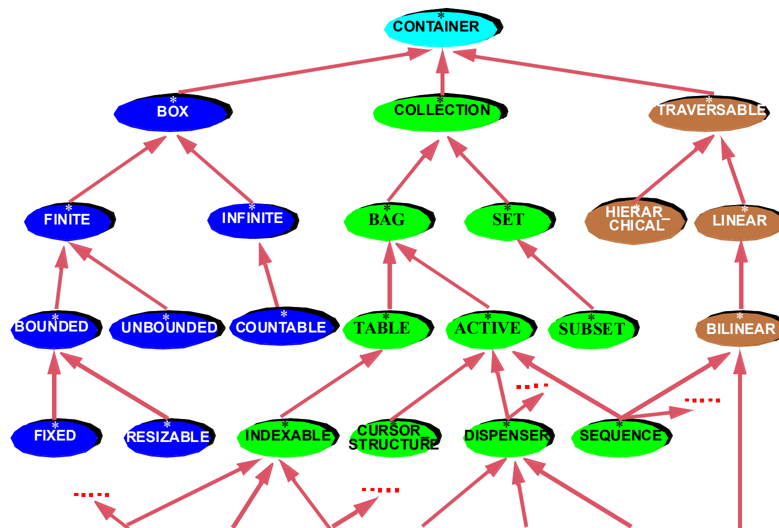
Background experience

Simula 67: O-O programming, a few reusable classes
Numerical libraries: IMSL, Linpack, NAG etc.
Design and implementation of the ISE Eiffel libraries:

- EiffelBase – fundamental data structures and algorithms. About 180 classes.
- EiffelVision – platform-independent graphics and GUI. About 550 classes.
- EiffelLex, EiffelParse – Lexical analysis and parsing of formal languages.
- EiffelStore – Platform-independent database access and support for persistent objects.
- EiffelMath: numerical computation.
- EiffelNet: Client-server object exchange.

(About 3000 classes, heavily reused in numerous applications.)
Also: application libraries developed in collaboration with customers.

Eiffelbase hierarchy



Simplicity

- Command-query separation principle:
  - Clear, understandable interfaces
- Systematic naming conventions
- Operand-option separation:
  - Dramatically simplified feature interfaces

7

The view from a traditional library (NAG)

```
nonlinear_ode
        (equation_count: in INTEGER;
         epsilon: in out DOUBLE;
         func: procedure
                   (eq_count: INTEGER; a: DOUBLE;
                    eps: DOUBLE; b: ARRAY [DOUBLE];
                    cm: pointer Libtype);
         left_count, coupled_count: INTEGER …)
```

[And so on. Altogether 19 arguments, including:

- 4 in out values;
- 3 arrays, used both as input and output;
- 6 functions, each with 6 or 7 arguments, of which 2 or 3 arrays!]

8

The EiffelMath routine

... Set up the non-default values ...

e.*solve*

... Solve the problem, recording the answer in *x* and *y* ...

9

Library design: the key issue

- Facilitating the process of learning to use a class

Steps:

- Finding out about the class
- Deciding whether it's useful
- Deciding which features are initially useful
- Learning to use these features

10

Library design

- The Formula-1 racing of software development
- Perfectionism is good!

11

Levels of reusability

- **0- Used in one system.**

1. **Used in several systems built by the same person.**
2. **Used in several systems built by the same group or company.**
3. **Used in several systems built by people that are in contact with the developers.**
4. **Used by groups unknown to the developers.**

12

The Consistency Principle

- All the components of a library should proceed from an overall coherent design, and follow a set of systematic, explicit and uniform conventions.

- Two components:
  - Top-down and deductive (the overall design).
  - Bottom-up and inductive (the conventions).

13

Abstraction and objects

- Not all classes describe “objects” in the sense of real-world things.
- Types of classes:
  - Analysis classes – examples: *AIRPLANE*, *CUSTOMER*, *PARTICLE*.
  - Design classes – examples: *STATE*, *COMMAND*, *HANDLE*.
  - Implementation classes – examples: *ARRAY*, *LINKED_LIST*.
- More important than the notion of object is the concept of abstract data type (or “data abstraction”).
- Key to the construction of a good library is the search for the best abstractions.

14

Avoiding improper classes

- A few danger signals:
  - A class whose name is a verb in the imperative form, e.g. *ANALYZE*. (Exception: command classes.)
  - A class with no parent and just one exported routine.
  - A class that introduces or redeclares no feature. (Purely taxonomical role only.) TAXOMANIA
- Names that warrant some attention: "er" names, e.g. *ANALYZER*.

Active data structures

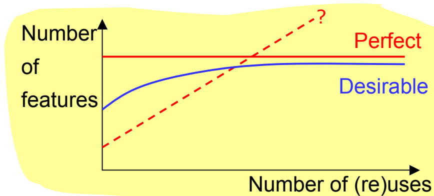
-- Typical sequence:

j := *l*.*search* (*x*);
l.*insert* (*j* + 1, *y*)

- Old interface for lists:

  l.*insert* (*i*, *x*)
  l.*remove* (*i*)
  pos := *l*.*search* (*x*)

  l.*insert_by_value* (…)
  l.*insert_by_position* (…)
  l.*search_by_position* (…)

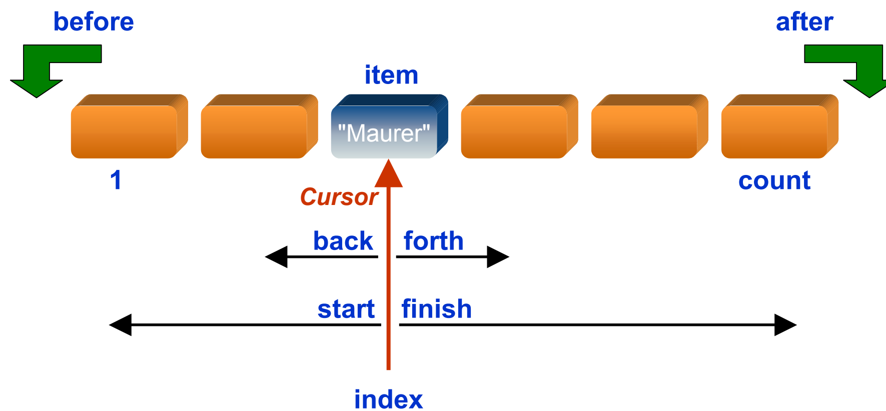


- New interface:

Queries:
l.*index* *l*.*item* *l*.*before* *l*.*after*

Commands:
l.*start* *l*.*forth* *l*.*finish* *l*.*back* *l*.*go* (*i*)
l.*search* (*x*)
l.*put* (*x*) l.remove

A list seen as an active data structure



17

An object as machine

18

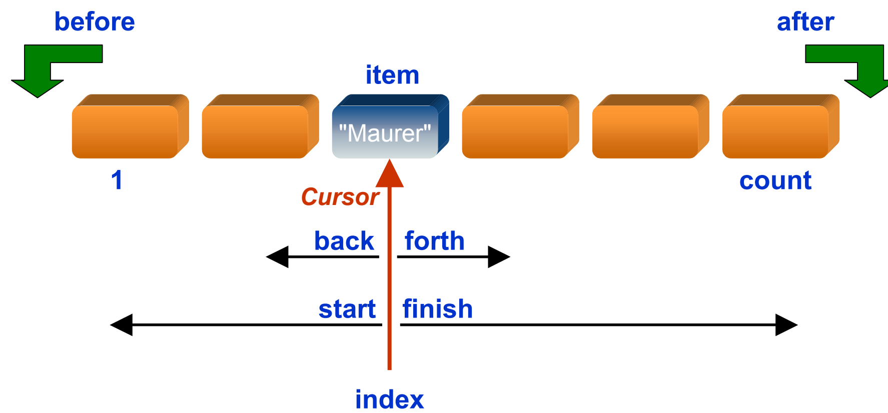
An object is a machine

19

An object is a machine

20

A list



21

An object has an interface

22

An object has an implementation

23

Information hiding

24

Command-Query separation principle

- Asking a question shouldn't change the answer

25

Command-Query separation principle

- A command (procedure) does something but does not return a result.
- A query (function or attribute) returns a result but does not change the state.

This principle excludes many common schemes, such as using functions for input (e.g. C's *getint* or equivalent).

26

Behind the principle: Referential Transparency

If two expressions have equal value, one may be substituted for the other in any context where that other is valid.

- If $a = b$, then $f(a) = f(b)$ for any f.
- Prohibits functions with side effects.
- Also:
  - For any integer i, normally $i + i = 2 \times i$;
  - But even if *getint* () = 2, *getint* () + *getint* () is usually not equal to 4.

27

Dijkstra (1968)

Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

28

Command-query separation

Input mechanism (instead of *n* := *getint* ()):

io.*read_integer*
n := *io*.*last_integer*

29

Would you rather buy or inherit?

- Inheritance is the "is-a" relation.
- In some cases "is-a" is clearly not applicable.
- Implementation can be a form of "is-a"
  - Example: the marriage of convenience.

30

When inheritance won't do

- From: Ian Sommerville: *Software Engineering*, 4th edition, Addison-Wesley:

  Multiple inheritance allows several objects to act as base objects and is supported in object-oriented languages such as Eiffel (Meyer, 1988). The characteristics of several different object classes can be combined to make up a new object.

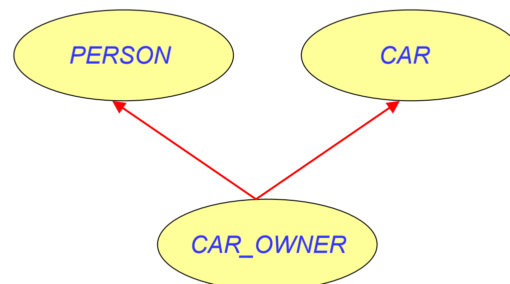
  For example, say we have an object class CAR which encapsulates information about cars and an object class PERSON which encapsulates information about people. We could use both of these to define a new object class CAR-OWNER which combines the attributes of CAR and PERSON.

  Adaptation through inheritance tends to lead to extra functionality being inherited, which can make components inefficient and bulky.

- Where is the "is-a"?

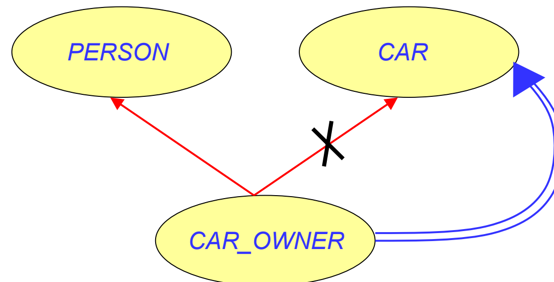
31

When inheritance won't do (cont'd)



32

The proper structure



The car-owner

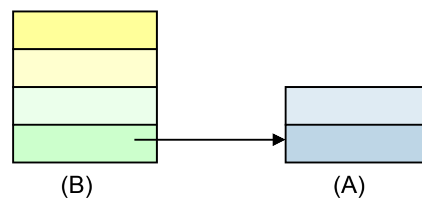
"He has a head like an Austin Mini with the doors open."

(From: *The Dictionary of Aussie Slang*,
Five-Mile Press, Melbourne, Australia.)

Would you rather buy or inherit?

Except for polymorphic uses, inheritance is never required:

- Rather than having *B* inherit from *A* you can always have *B* include an attribute of type *A* (or **expanded** *A*) – except if an entity of type *A* may have to represent values of type *B*.



35

To be is also to have!

- (1) Every software engineer is an engineer.
- (2) Every software engineer has a part of himself which is an engineer.

But:

TO HAVE IS NOT ALWAYS TO BE!

36

Would you rather buy or inherit?

A case in which having is not being (i.e. "client" is OK but not inheritance):

- Every object of type *B* has a component of type *A*, BUT that component may need to be replaced during the object's lifetime.

Use the client relation instead:

```
class WINDOW inherit
        GENERAL_WINDOW

        WINDOW_IMPLEMENTATION
feature
        ...
end
```

37

Handles

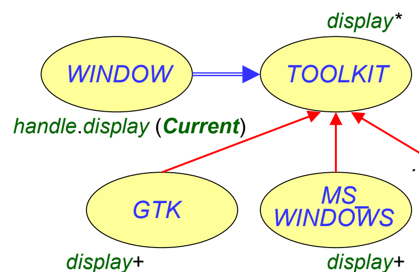
```
class WINDOW inherit

    GENERAL_WINDOW

feature

    handle: TOOLKIT
    ...
    set_handle (t: TOOLKIT) is
        do
            handle := t
        end
    ...
end
```

```
display is
    do
        handle.display (Current)
    end
```



38

Handles (continued)

```
class TOOLKIT_FACILITIES feature

    impl: IMPLEMENTATION is
            once
                    create Result
            end

    set_handle (t: TOOLKIT) is
            do
                    impl.set_handle (t)
            end

end
```

This is a class meant to be inherited by classes needing its facilities.

39