Principles of Library Design: The Eiffel Experience

Bertrand Meyer

ADFOCS Summer School, 2003

LECTURE 2

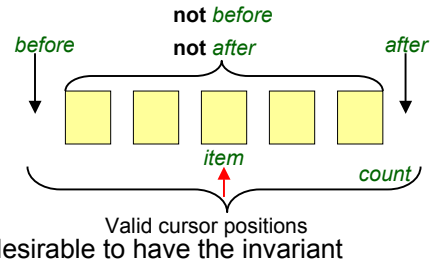
"Plan"

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

Designing for consistency: An example

- Describing active structures properly: can after also be before?
- Symmetry:

start	*finish*
forth	*back*
after	*before*



- For symmetry and consistency, it is desirable to have the invariant properties.

A
- *after* = (*index* = *count* + 1)
- *before* = (*index* = 0)

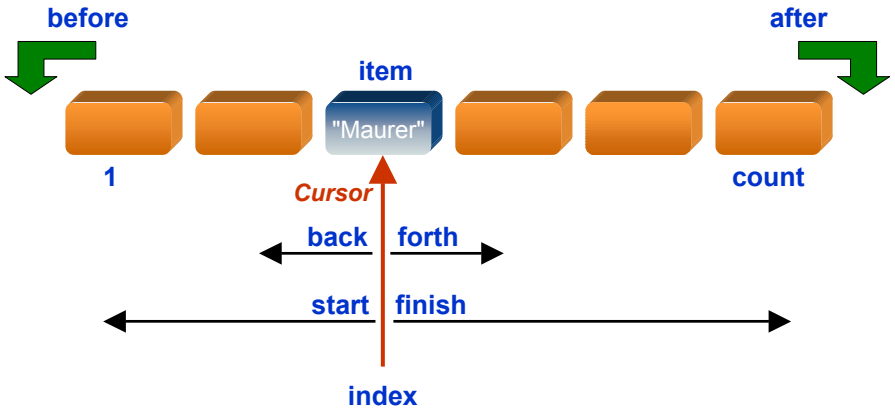
Designing for consistency (continued)

- Typical iteration:

```
from
        start
until
        after
loop
        some_action (item)
        forth
end
```

- Conventions for an empty structure?
  - *after* must be true for the iteration.
  - For symmetry: *before* should be true too.
- But this does not work for an empty structure (*count* = 0, see invariant A): should *index* be 0 or 1?

A list



5

Designing for consistency (continued)

To obtain a consistent convention we may transform the invariant into:

B
after = (*is_empty* **or** (*index* = *count* + 1))
before = (*is_empty* **or** (*index* = 0)
-- Hence: *is_empty* = (*before* and *after*)

Symmetric but unpleasant. Leads to frequent tests of the form

if *after* and **not** *is_empty* **then** ...

instead of just

if *after* **then** ...

6

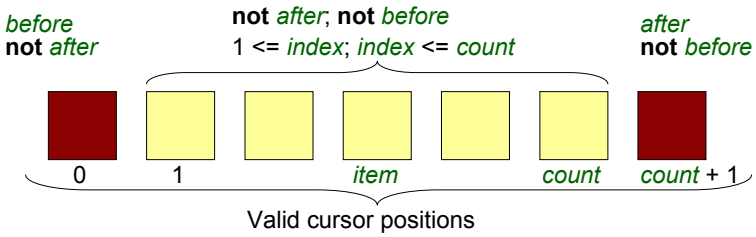
Introducing sentinel items

- Invariant (partial):

  0 <= *index*

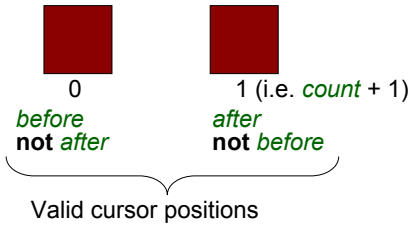
  index <= *count* + 1

  A { *before* = (*index* = 0)
  A { *after* = (*index* = *count* + 1)
  A { **not** (*after* **and** *before*)



7

The case of an empty structure



8

Can after also be before?

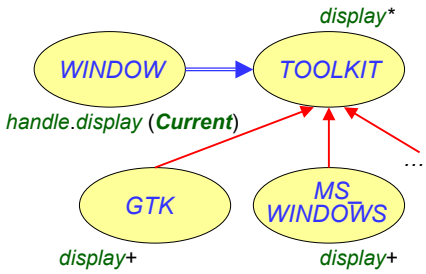
Lessons from an example; General principles:

- Consistency
  - A posteriori: “How do I make this design decision compatible with the previous ones?”.
  - A priori: “How do I take this design decision so that it will be easy – or at least possible – to make future ones compatible with it?”.
- Use assertions, especially invariants, to clarify the issues.
- Importance of symmetry concerns (cf. physics and mathematics).
- Importance of limit cases (empty or full structures).

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
```



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.

11

How big should a class be?

The first question is how to measure class size. Candidate metrics:

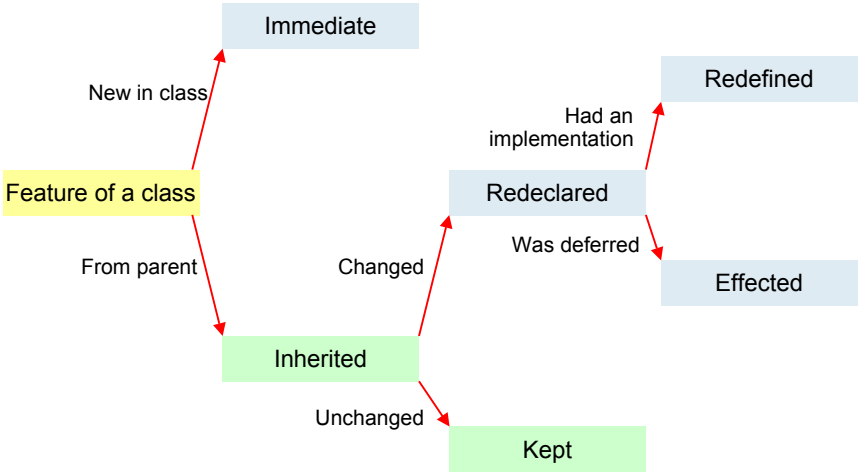
- Source lines.
- Number of features.

For the number of features the choices are:

- With respect to information hiding:
  - Internal size: includes non-exported features.
  - External size: includes exported features only.
- With respect to inheritance:
  - Immediate size: includes new (immediate) features only.
  - Flat size: includes immediate and inherited features.
  - Incremental size: includes immediate and redeclared features.

12

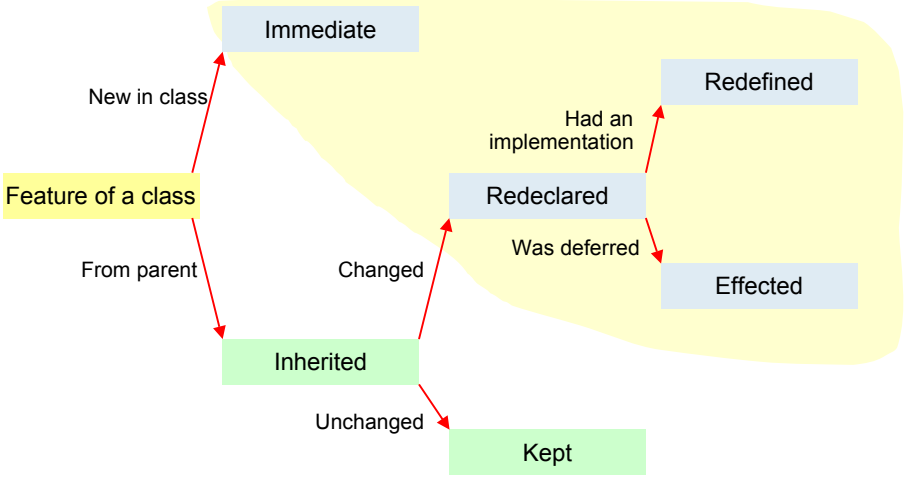
The features of a class



Most useful measure is incremental size. Easy to measure.

13

The features of a class



Most useful measure is incremental size. Easy to measure.

14

The shopping list approach

- If a feature may be useful, it probably is.
- An extra feature cannot hurt if it is designed according to the spirit of the class (i.e. properly belongs in the underlying abstract data type), is consistent with its other features, and follows the principles of this presentation.
- No need to limit classes to “atomic” features.

15

Some statistics from EiffelBase

- Percentages, rounded. (Includes EiffelLex and EiffelParse.)
- 149 classes, 1823 exported features.

0 to 5 features	45
6 to 10 features	17
11 to 15 features	11
16 to 20 features	9
21 to 40 features	13
41 to 80 features	4
81 to 142 features	1

16

Some statistics from EiffelVision

- Percentages, rounded. 546 classes, 3666 exported features.

0 to 5 features	68
6 to 10 features	12
11 to 15 features	7
16 to 20 features	4
21 to 40 features	6
41 to 78 features	2

17

Including non-exported features

- Percentage rounded. All features (about 7600).

	Base	Vision
0 to 5 features	37	55
6 to 10 features	23	18
11 to 15 features	7	7
16 to 20 features	6	5
21 to 40 features	16	10
41 to 80 features	9	4
81 or more features	2	0.4

- Ratio of total features to exported features: 1.27 (EiffelBase), 1.44 (EiffelVision)

18

Minimalism revisited

The language should be small (ETL: *"The language design should provide a good way to express every operation of interest; it should avoid providing two."*)

The library, in contrast, should provide as many useful facilities as possible.

Key to a non-minimalist library:

- Consistent design.
- Naming.
- Contracts.

Usefulness and power.

19

The size of feature interfaces

- More relevant than class size for assessing complexity.
- Statistics from EiffelBase, EiffelLex and EiffelParse (exported features):

Number of features	1823
Percentage of queries	59%
Percentage of commands	41%
Average number of arguments to a feature	0.4
Maximum number	3
No argument	60%
One argument	37%
Two arguments	3%
Three arguments	0.3%

20

The size of feature interfaces (cont'd)

Including non-exported features:

Average number of arguments to a feature	0.5
Maximum number	6
No argument	57%
One argument	36%
Two arguments	5%
Three arguments	1%
Four arguments	0.6%
Five or six arguments	0.2%

21

The size of feature interfaces (cont'd)

- Statistics from EiffelVision 1 (546 classes, exported features):

Number of features	3666
Percentage of queries	39%
Percentage of commands	61%
Average number of arguments to a feature	0.7
Maximum number	7
No argument	49%
One arguments	32%
Two arguments	15%
Three arguments	3%
Four arguments	0.4%
Five to seven arguments	0.4%

22

Operands and options

Two possible kinds of argument to a feature:

- Operands: values on which feature will operate.
- Options: modes that govern how feature will operate.

Example: printing a real. The number is an operand; format properties (e.g. number of significant digits, width) are options.

print (*real_value*, *number_of_significant_digits*,
zone_length, *number_of_exponent_digits*, ...)

my_window.display (*x_position*, *y_position*,
height, *width*, *text*, *title_bar_text*, *color*, ...)

23

Recognizing options from operands

Two criteria to recognize an option:

- There is a reasonable default value.
- During the evolution of a class, operands will normally remain the same, but options may be added.

24

Operands and options

The Option Principle:

- The arguments of a feature should only be operands.

Options should have default values, with procedures to set different values if requested.

For example:

```
my_window.set_background_color ("blue")
...
my_window.display
```

25

Operands and options

- Useful checklist for options:

Option	Default	Set	Accessed
Window color	White	*set_background_color*	*background_color*
Hidden?	No	*set_visible* *set_hidden*	*hidden*

26

Naming (1)

Class	Features		
ARRAY	*enter*	*entry*	
STACK	*push*	*top*	*pop*
QUEUE	*add*	*oldest*	*remove_oldest*
HASH_TABLE	*insert*	*value*	*delete*

27

Naming (2)

Class	Features		
ARRAY	*put*	*item*	
STACK	*put*	*item*	*remove*
QUEUE	*put*	*item*	*remove*
HASH_TABLE	*put*	*item*	*remove*

28

Naming rules

Achieve consistency by systematically using a set of standardized names.

Emphasize commonality over differences.

Differences will be captured by:

- Signatures (number and types of arguments and result).
- Assertions.
- Comments.

29

Some standard names

Queries:

count
item, **infix** "@"
to_external, *to_c*, *from_external*

-- Array access:
a.item (*i*) or *a* @ *i*

Commands:

make -- For creation
put, *extend*, *replace*, *force*
remove, *prune*, *wipe_out*

-- Rejected names:
if *s.addable* **then**
    s.add (*v*)
end
if *s.deletable* **then**
    s.delete (*v*)
end

Boolean queries:

writable, *readable*, *extendible*, *prunable*
empty, *full*
capacity

-- Usual invariants:
-- empty = (count = 0)
-- full = (count = capacity)

30

Grammatical rules

- Procedures (commands): verbs, infinitive form. Examples: *make*, *put*, *display*.
- Boolean queries: adjectives, e.g. *full*. Also (especially in case of potential ambiguity) names of the form *is_some_property*. Example: *is_first*.
  - In all cases, you should usually choose the form of the property that is false by default at initialization (making it true is an event worth talking about). Example: *is_erroneous*.
- Other queries: nouns or adjectives.Examples: *count*, *error_ window*.
- Do not use verbs for queries, in particular functions; this goes with the command-query separation principle (prohibition of side-effects in functions).

31

Feature categories

```
class C inherit
        …
feature -- Category 1

        … Feature declarations

feature {A, B} -- Category 2

        … Feature declarations

feature {NONE} -- Category n

        … Feature declarations

invariant
        …
end
```

32

Feature categories (cont'd)

- Standard categories (the only ones in EiffelBase):
  - Initialization
  - Access
  - Measurement
  - Comparison
  - Status report
  - Status setting
  - Cursor movement
  - Element change
  - Removal
  - Resizing
  - Transformation
  - Conversion
  - Duplication
  - Basic operations
  - Obsolete
  - Inapplicable
  - Implementation
  - Miscellaneous

33

Obsolete features and classes

A central problem in IT: how to reconcile progress with the protection of the installed base?

Obsolete features and classes support smooth evolution.

In class *ARRAY:*

```
enter (i: V; v: T) is
        obsolete "Use `put (value, index)'"
        do
                put (v, i)
        end
```

34

Obsolete classes

```
class
    ARRAY_LIST [G]
obsolete
    "[
        Use MULTI_ARRAY_LIST instead
        (same semantics, but new name
        ensures more consistent terminology).
        Caution: do not confuse with ARRAYED_LIST
        (lists implemented by one array each).
    ]"
inherit

    MULTI_ARRAY_LIST [G]

end
```

35

Eiffel

- Method, language, environment
- Object-oriented to the core
- Design by Contract
- Soon an ECMA standard
- Used in mission-critical systems worldwide
- Lots of platforms
- Closely integrated with .NET
- Also a key tool for education

36

The Eiffel method: some principles

- Abstraction
- Information hiding
- Seamlessness
- Reversibility
- Design by Contract
- Open-Closed principle
- Single choice principle
- Single model principle
- Uniform access principle
- Command-query separation principle
- Option-operand separation principle
- Style matters

... See next...

37

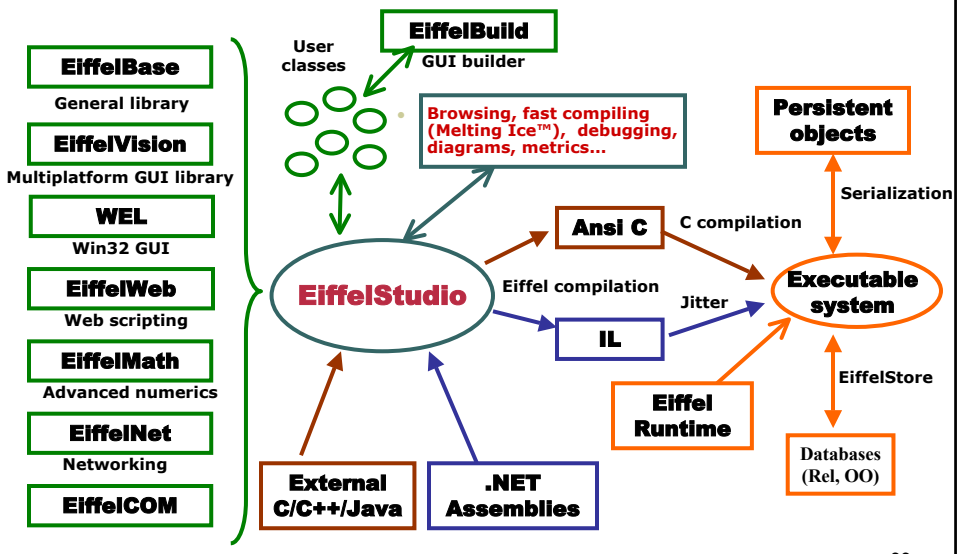
Environment: the two offerings from Eiffel Software

- EiffelStudio ("Classic Eiffel")
  Windows, Unix, Linux, VMS, .NET ...
- ENViSioN! for Visual Studio .NET

Projects are compatible

38

EiffelStudio



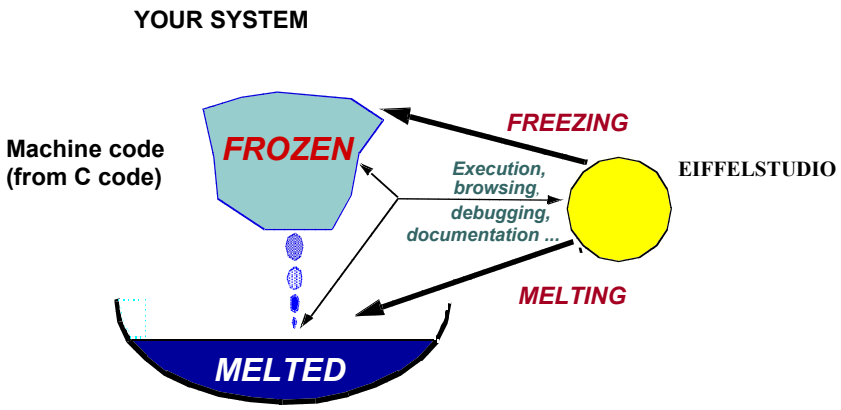
39

EiffelStudio: Melting Ice™ Technology

- Fast recompilation: time depends on size of change, not size of program
- Full type checking
- "Freeze" once in a while
- Optimized compilation: finalize.

40

Melting Ice Technology



41

Eiffel for .NET

- One of the first languages to be available for .NET, right from the time of first announcement
- Full language, with multiple inheritance, Design by Contract, genericity etc.
- Full player: interoperability, consumer, producer, extender, verifiable, CLS-compliant...
- Choice between EiffelStudio and Visual Studio (ENViSioN!)

42

Some Eiffel Software users

AXA Rosenberg

Boeing

Chicago Board of Trade

AMP Investments

EMC

Lockheed Martin

Environmental Protection Agency

Hewlett Packard

Cap Gemini Ernst & Young

Swedish National Health Board

ENEA

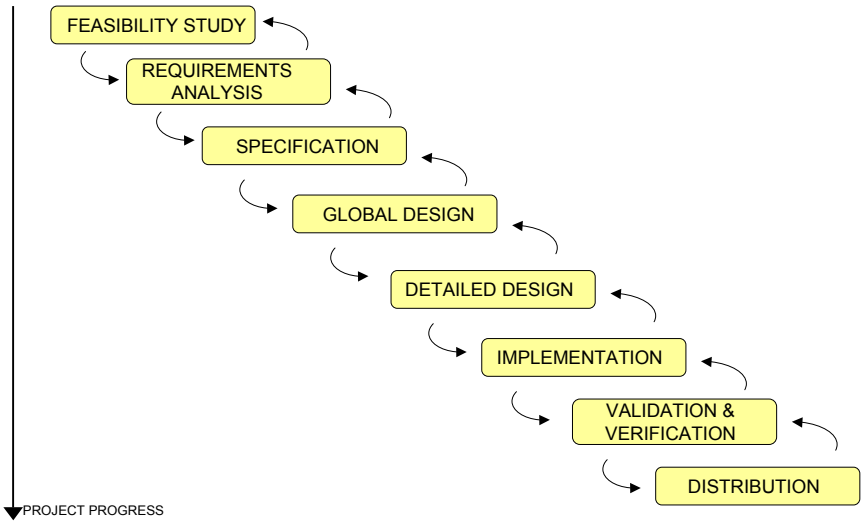
CALFP

Levels of reusability

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

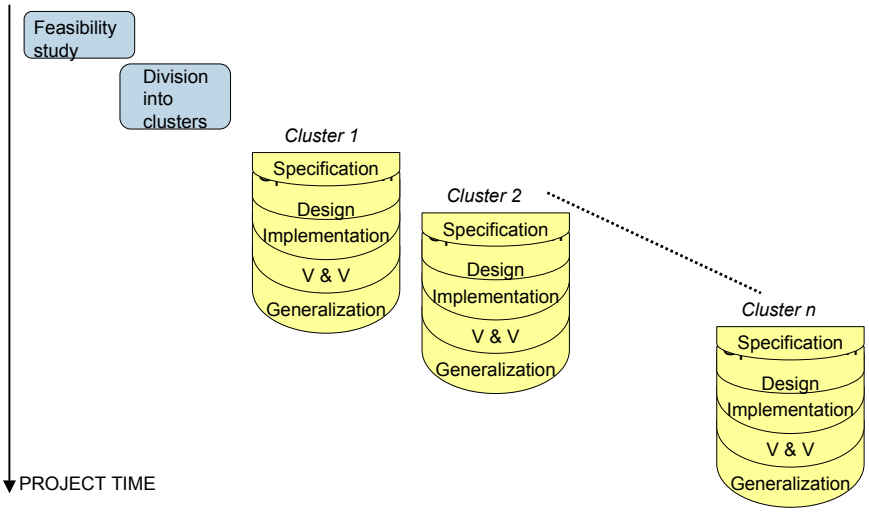
1. **Used in several systems built by the same person.**
2. **Used in several systems built by the same group or company.**
3. **Used in several systems built by people that are in contact with the developers.**
4. **Used by groups unknown to the developers.**

The waterfall model of the lifecycle



45

The cluster model



46

Development: the traditional view

Separate tools:

- Programming environment
- Analysis & design tools, e.g. UML

Consequences:

- Hard to keep model, implementation, documentation consistent
- Constantly reconciling views
- Inflexible, hard to maintain systems
- Hard to accommodate bouts of late wisdom
- Wastes efforts
- Damages quality

47

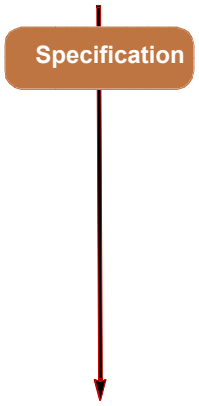
Development: the Eiffel view

Seamless development:

- Single set of notation, tools, concepts, principles throughout
- Eiffel is as much for analysis & design as for implementation & maintenance
- Continuous, incremental development
- Keep model, implementation and documentation consistent
- Reversibility: can go back and forth
- Saves money: invest in single set of tools
- Boosts quality

48

Seamless development (1)

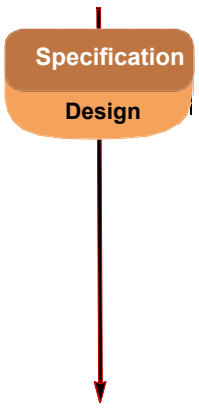


TRANSACTION, PLANE,
CUSTOMER, ENGINE...

Example classes

49

Seamless development (2)



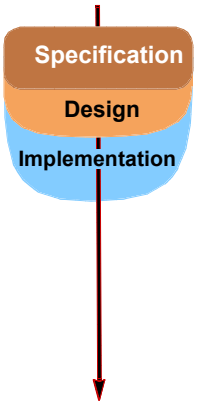
TRANSACTION, PLANE,
CUSTOMER, ENGINE...

STATE, USER_COMMAND...

Example classes

50

Seamless development (3)



TRANSACTION, PLANE, CUSTOMER, ENGINE...

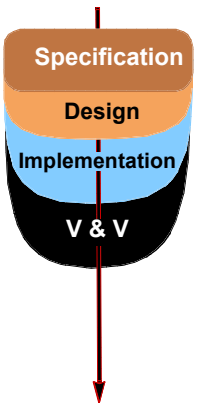
STATE, USER_COMMAND...

HASH_TABLE, LINKED_LIST...

Example classes

51

Seamless development (4)



TRANSACTION, PLANE, CUSTOMER, ENGINE...

STATE, USER_COMMAND...

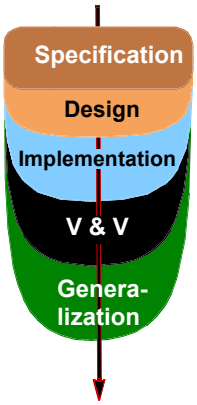
HASH_TABLE, LINKED_LIST...

TEST_DRIVER, ...

Example classes

52

Seamless development (5)



TRANSACTION, PLANE,
CUSTOMER, ENGINE...

STATE, USER_COMMAND...

HASH_TABLE,
LINKED_LIST...

TEST_DRIVER, ...

AIRCRAFT, ...

Example classes

Eiffel for analysis

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                    -- Fill the vat.
            require
                    in_valve.open
                    out_valve.closed
            deferred
            ensure
                    in_valve.closed
                    out_valve.closed
                    is_full
            end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant
    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

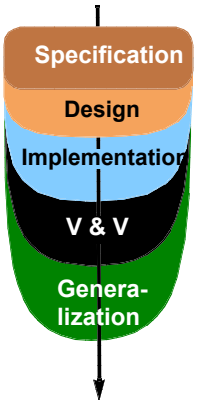
Precondition

-- Specified only.
-- not implemented.

Postcondition

Class invariant

Seamless development



TRANSACTION, PLANE,
CUSTOMER, ENGINE...

STATE, USER_COMMAND...

HASH_TABLE,
LINKED_LIST...

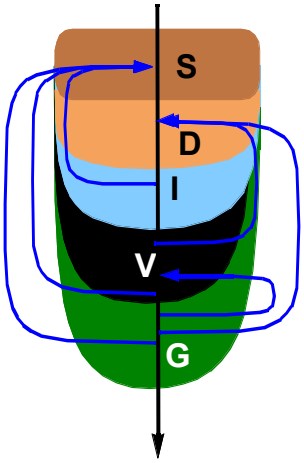
TEST_DRIVER, ...

AIRCRAFT, ...

Example classes

55

Reversibility



56

EiffelStudio support for seamless development

Diagram Tool

- System diagrams can be produced automatically from software text
- Works both ways: update diagrams or update text – other view immediately updated
- No need for separate UML tool

Metrics Tool

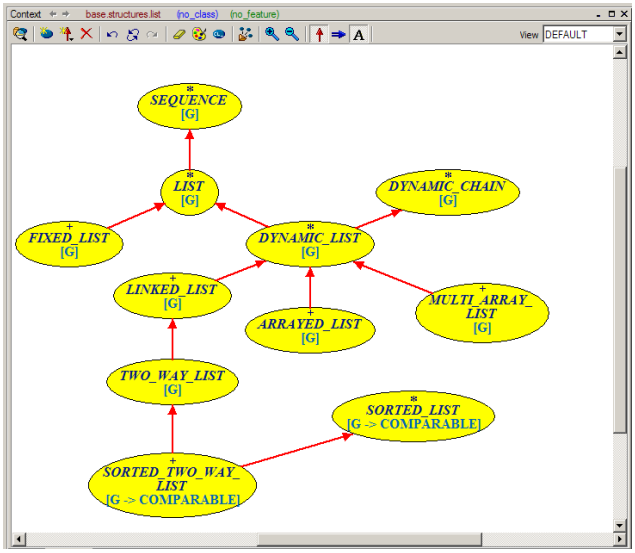
Profiler Tool

Documentation generation tool

...

57

Inheritance structure (in EiffelStudio)



58

Design by Contract™

- Get things right in the first place
- Automatic documentation
- Self-debugging, self-testing code
- Get inheritance right
- Give managers the right control tools

59

Applications of contracts

- Analysis, design, implementation:
  Get the software right from the start

- Testing, debugging, quality assurance
- Management, maintenance/evolution
- Inheritance
- Documentation

60

A human contract

deliver	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 a.m. next day.	(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.

61

Properties of contracts

A contract:

- Binds two parties (or more): supplier, client.
- Is explicit (written).
- Specifies mutual obligations and benefits.
- Usually maps obligation for one of the parties into benefit for the other, and conversely.
- Has no hidden clauses: obligations are those specified.
- Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices).

62

A view of software construction

- Constructing systems as structured collections of cooperating software elements — suppliers and clients — cooperating on the basis of clear definitions of obligations and benefits.
- These definitions are the contracts.

63

Contracts for analysis

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                -- Fill the vat.
        require
                in_valve.open
                out_valve.closed
        deferred
        ensure
                in_valve.closed
                out_valve.closed
                is_full
        end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant
    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

Precondition

-- Specified only.

-- not implemented.

Postcondition

Class invariant

64

Contracts for analysis

fill	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure input valve is open, output valve is closed.	(From postcondition:) Get filled-up vat, with both valves closed.
Supplier	(Satisfy postcondition:) Fill the vat and close both valves.	(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position.

65

Correctness in software

Correctness is a relative notion: consistency of implementation vis-à-vis specification. (This assumes there is a specification!)

Basic notation: (*P*, *Q*: assertions, i.e. properties of the state of the computation. *A*: instructions).

$$\{P\}\ A\ \{Q\}$$

"Hoare triple"

What this means (total correctness):

- Any execution of *A* started in a state satisfying *P* will terminate in a state satisfying *Q*.

66

Hoare triples: a simple example

{n > 5} *n* := *n* + 9 {n > 13}

Most interesting properties:

- *Strongest* postcondition (from given precondition).
- *Weakest* precondition (from given postcondition).

"*P* is stronger than or equal to *Q*" means:
P implies *Q*

- QUIZ: What is the strongest possible assertion? The weakest?

67

Specifying a square root routine

{*x* >= 0}

... Square root algorithm to compute *y* ...

{*abs* (*y* ^ 2 – *x*) <= 2 * *epsilon* * *y*}
-- i.e.: *y* approximates exact square root of *x*
-- within *epsilon*

68

Software correctness

Consider

{*P*} *A* {*Q*}

Take this as a job ad in the classifieds.

Should a lazy employment candidate hope for a weak or strong *P*? What about *Q*?

Two special offers:

- 1. {*False*} *A* {...}
- 2. {...} *A* {*True*}

69

A contract (from EiffelBase)

```
extend (new: G; key: H)
            -- Assuming there is no item of key key,
            -- insert new with key; set inserted.
        require
            key_not_present: not has (key)
        ensure
            insertion_done: item (key) = new
            key_present: has (key)
            inserted: inserted
            one_more: count = old count + 1
```

70

The contract

Routine	OBLIGATIONS	BENEFITS
Client	PRECONDITION	POSTCONDITION
Supplier	POSTCONDITION	PRECONDITION

71

A class without contracts

```
class ACCOUNT  feature -- Access

    balance: INTEGER
                -- Balance

    Minimum_balance: INTEGER is 1000
        -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

    add (sum: INTEGER) is
                -- Add sum to the balance (secret procedure).
        do
                balance := balance + sum
        end
```

72

Without contracts (cont'd)

```
feature -- Deposit and withdrawal operations

    deposit (sum: INTEGER) is
                -- Deposit sum into the account.
        do
                add (sum)
        end

    withdraw (sum: INTEGER) is
                -- Withdraw sum from the account.
        do
                add (– sum)
        end

    may_withdraw (sum: INTEGER): BOOLEAN is
                -- Is it permitted to withdraw sum from the account?
        do
                Result := (balance - sum >= Minimum_balance)
        end
end
```

73

Introducing contracts

```
class ACCOUNT create

    make

feature {NONE} -- Initialization

    make (initial_amount: INTEGER) is
                -- Set up account with initial_amount.

        require
                large_enough: initial_amount >= Minimum_balance
        do
                balance := initial_amount
        ensure
                balance_set: balance = initial_amount
        end
```

74

Introducing contracts (continued)

```
feature -- Access

    balance: INTEGER
                -- Balance

    Minimum_balance: INTEGER is 1000
            -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

    add (sum: INTEGER) is
                -- Add sum to the balance (secret procedure).
            do
                balance := balance + sum
            ensure
                increased: balance = old balance + sum
            end
```

75

With contracts (cont'd)

```
feature -- Deposit and withdrawal operations

    deposit (sum: INTEGER) is
                -- Deposit sum into the account.
            require
                not_too_small: sum >= 0
            do
                add (sum)
            ensure
                increased: balance = old balance + sum
            end
```

76

With contracts (cont'd)

```
withdraw (sum: INTEGER) is
        -- Withdraw sum from the account.
    require
        not_too_small: sum >= 0
        not_too_big:  sum <= balance – Minimum_balance
    do
        add (– sum)
            -- i.e. balance := balance – sum
    ensure
        decreased: balance = old balance - sum
    end
```

77

The contract

withdraw	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure *sum* is neither too small nor too big.	(From postcondition:) Get account updated with *sum* withdrawn.
Supplier	(Satisfy postcondition:) Update account for withdrawal of *sum*.	(From precondition:) Simpler processing: may assume *sum* is within allowable bounds.

78

The imperative and the applicative

do *balance* := *balance* - *sum*	**ensure** *balance* = **old** *balance* - *sum*
PRESCRIPTIVE	DESCRIPTIVE
How?	What?
Operational	Denotational
Implementation	Specification
Command	Query
Instruction	Expression
Imperative	Applicative

79

On the Autobahn:

- Licht!
- *Licht ?*

80

With contracts (end)

```
may_withdraw (sum: INTEGER): BOOLEAN is
                -- Is it permitted to withdraw sum from the
                -- account?
        do
                Result := (balance - sum >= Minimum_balance)
        end

invariant
    not_under_minimum: balance >= Minimum_balance
end
```

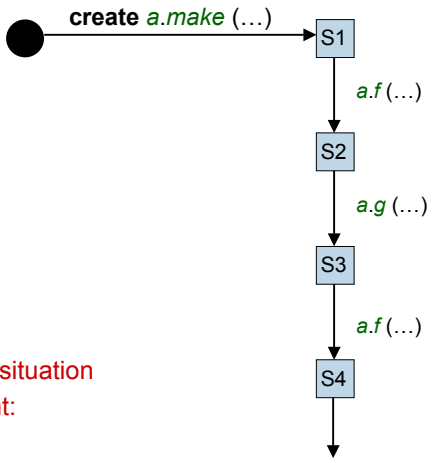
81

The class invariant

- Consistency constraint applicable to all instances of a class.
- Must be satisfied:
  - After creation.
  - After execution of any feature by any client.
    (Qualified calls only: *a.f* (...))

82

The correctness of a class



- For every creation procedure *cp*:

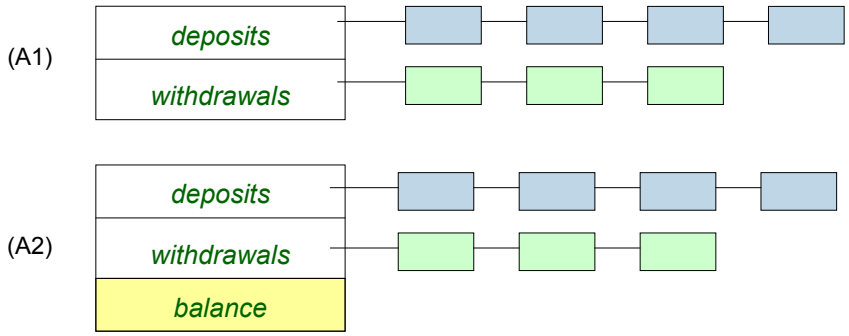
  $\{pre_{cp}\}\ do_{cp}\ \{post_{cp}\ \text{and INV}\}$

- For every exported routine *r*:

  $\{\text{INV and } pre_{r}\}\ do_{r}\ \{post_{r}\ \text{and INV}\}$

- The worst possible erroneous run-time situation in object-oriented software development:
  - Producing an object that does not satisfy the invariant of its own class.

The Uniform Access Principle



balance = *deposits.total* – *withdrawals.total*

EiffelStudio documentation

- Eiffel projects benefit from richest documentation, produced automatically from the class text
- Available in text, HTML, Postscript, RTF, FrameMaker and many other formats
- Numerous views, textual and graphical

85

Contracts as automatic documentation

Demo

LINKED_LIST Documentation,
generated by EiffelStudio

86

Contracts for analysis

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                -- Fill the vat.
        require
                in_valve.open
                out_valve.closed
        deferred
        ensure
                in_valve.closed
                out_valve.closed
                is_full
        end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant
    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

Precondition

-- Specified only.
-- not implemented.

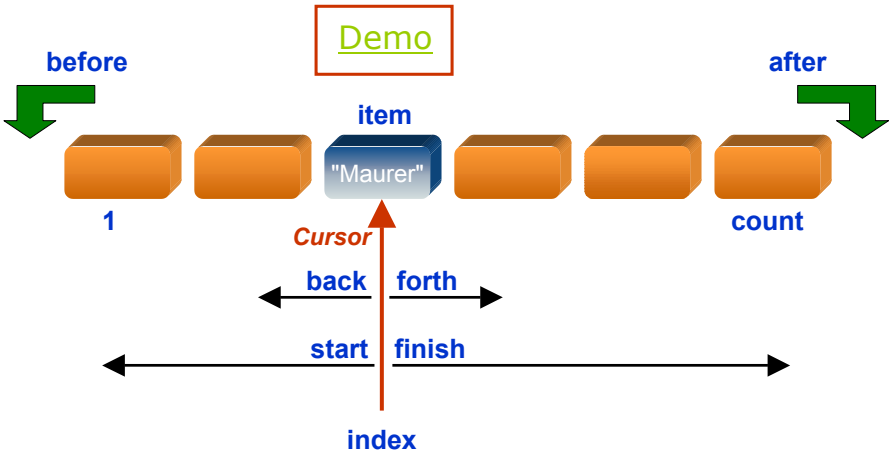
Postcondition

Class invariant

Contracts for testing and debugging

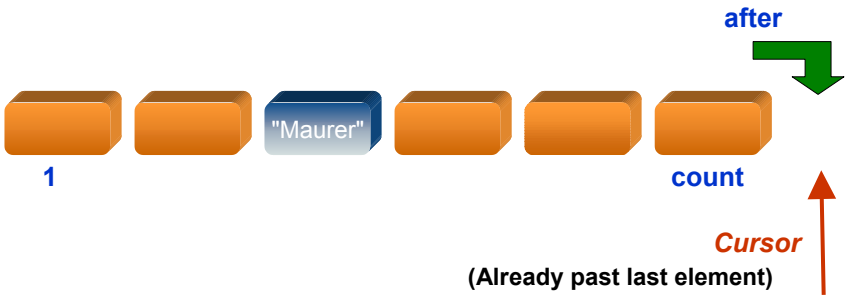
- Contracts express implicit assumptions behind code
- A bug is a discrepancy between intent and code
- Contracts state the intent!
- In EiffelStudio: select compilation option for run-time contract monitoring. Can be set a system, cluster, class level.
- May disable monitoring when releasing software
- A revolutionary form of quality assurance

Example: inserting into a list



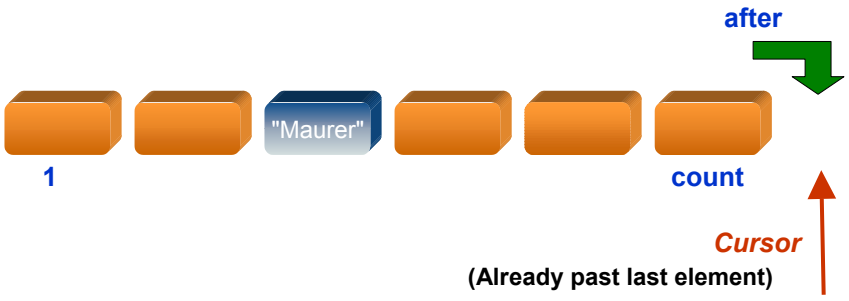
89

Trying to insert too far right



90

Can't insert!



91

Query "before" and command "forth"

Launch

Editor

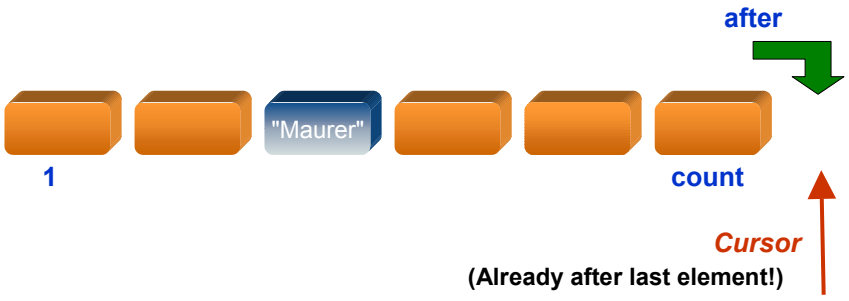
```
    before: BOOLEAN is
            -- Is there no valid cursor position to the left of curso
        do
            Result := (index = 0)
        end

feature -- Cursor movement

    forth is
            -- Move to next position; if no next position,
            -- ensure that `exhausted' will be true.
        deferred
        ensure then
            moved_forth: index = old index + 1
        end
```

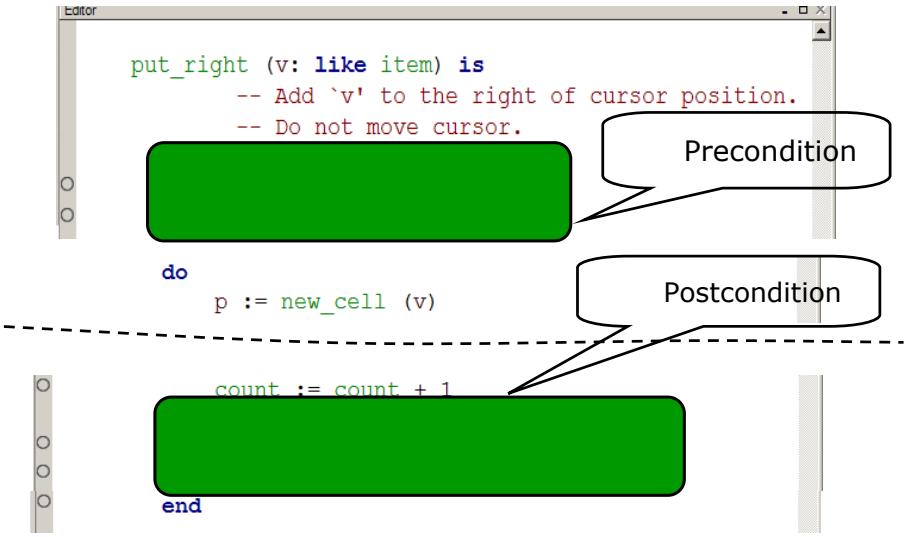
92

Can't insert!



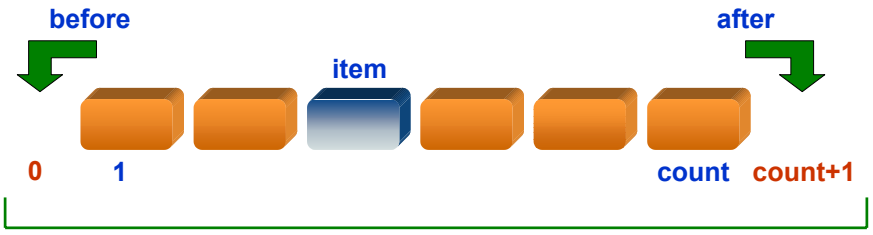
93

A command and its contract



94

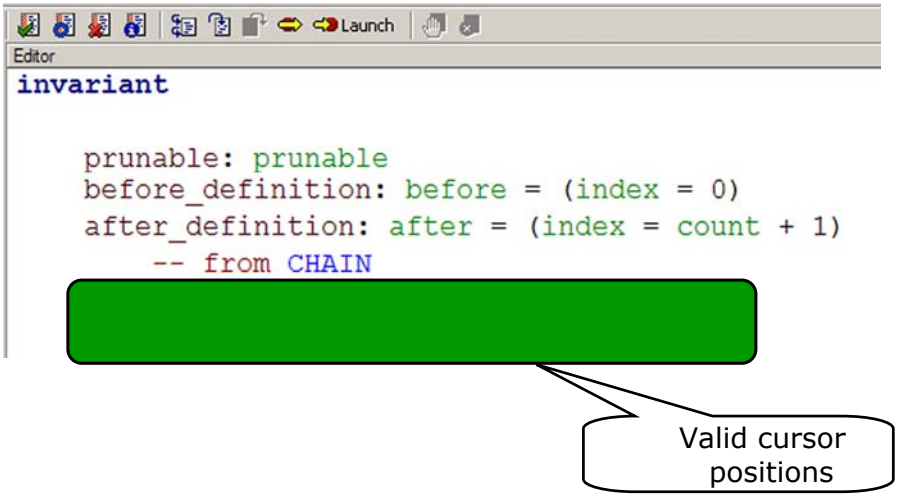
Where the cursor may go



Valid cursor positions

95

From the invariant of class LIST



96

Contract monitoring

A contract violation always signals a bug:

- Precondition violation: bug in client
- Postcondition violation: bug in routine

97

Contracts and inheritance

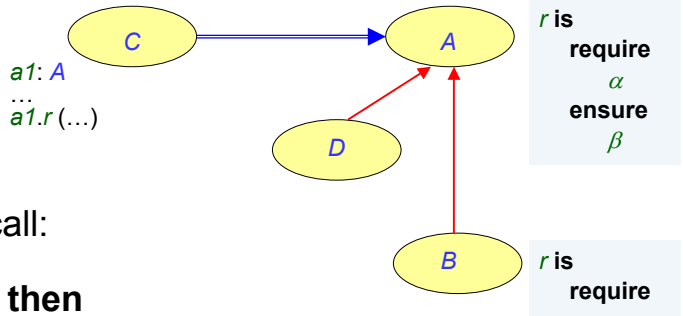
- Issues: what happens, under inheritance, to
  - Class invariants?
  - Routine preconditions and postconditions?

98

Invariants

- Invariant Inheritance rule:
  - The invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed.
- Accumulated result visible in flat and interface forms.

Contracts and inheritance



Correct call:

if *a1*.α **then**
    a1.r (...)
    -- Here *a1*.β holds.
end

Assertion redeclaration rule

- When redeclaring a routine:
  - Precondition may only be kept or weakened.
  - Postcondition may only be kept or strengthened.

101

Assertion redeclaration rule in Eiffel

- A simple language rule does the trick!
- Redefined version may have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

- Resulting assertions are:
  - *original_precondition* **or** *new_pre*
  - *original_postcondition* **and** *new_post*

102