

Last update: 10 April 2007

# Software Architecture

## Bertrand Meyer

ETH Zurich, March-July 2007

### Lecture 4: Design by Contract

## Applications

- Getting the software right
- Analysis
- Design
- Implementation
- Debugging
- Testing
- Management
- Maintenance
- Documentation

4

## Reading material

**OOSC2:**

- Chapter 11: Design by Contract

2

## Design by Contract

- Origin: work on "axiomatic semantics" (Floyd, Hoare, Dijkstra), early seventies
- Some research languages had a built-in assertion mechanism: Euclid, Alphas
- Eiffel introduced the connection with object-oriented programming and made contracts a software construction methodology and an integral part of the language
- Mechanisms for other languages: Nana macro package for C++, JML for Java, Spec# (and dozens of others)

5

## Design by Contract

A discipline of analysis, design, implementation, management

3

## Documentation Issues

Who will do the program documentation (technical writers, developers) ?

How to ensure that it doesn't diverge from the code (the French driver's license / reverse Dorian Gray syndrome) ?

**The Single Product principle**

The product is the software

6

## Design by Contract

Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users).

This goal is the element's **contract**.

The contract of any software element should be

- **Explicit.**
- **Part of the software element itself.**



7

## Ariane 5 (Conclusion)

The main lesson:

Reuse without a contract is sheer folly

See:

Jean-Marc Jézéquel and Bertrand Meyer

*Design by Contract: The Lessons of Ariane*

IEEE Computer, January 1997

Also at <http://www.eiffel.com>



10

## Ariane 5, 1996

\$500 million, not insured.

37 seconds into flight, exception in Ada program not processed; order given to abort the mission.

Exception was caused by an incorrect conversion: a 64-bit real value was incorrectly translated into a 16-bit integer.

- Not a design error.
- Not an implementation error.
- Not a language issue.
- Not really a testing problem.
- Only partly a quality assurance issue.

Systematic analysis had "proved" that the exception could not occur – the 64-bit value ("horizontal bias" of the flight) was proved to be always representable as a 16-bit integer !



8

## A human contract

	OBLIGATIONS	BENEFITS
<i>deliver</i> <i>Client</i>	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
<i>Supplier</i>	(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.



11

## Ariane 5(Continued)

It was a REUSE error:

- The analysis was correct – for Ariane 4 !
- The assumption was documented – in a design document !

With assertions, the error would almost certainly (if not avoided in the first place) detected by either static inspection or testing:

```
integer_bias (b: REAL): INTEGER is
  require
    representable (b)
  do
    ...
  ensure
    equivalent (b, Result)
end
```



9

## 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.



12

## 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).

13

## So, is it like "assert.h"?

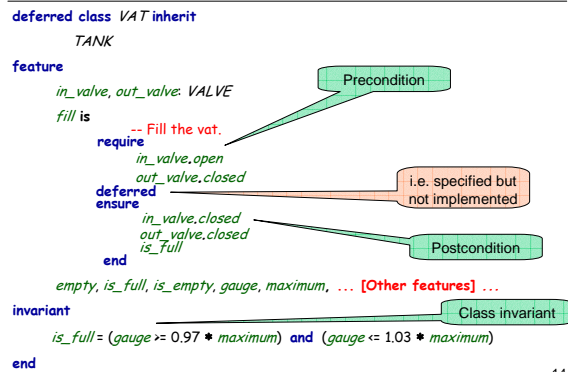
(Source: Reto Kramer)

### Design by Contract goes further:

- "Assert" does not provide a contract.
- Clients cannot see asserts as part of the interface.
- Asserts do not have associated semantic specifications.
- Not explicit whether an assert represents a precondition, post-conditions or invariant.
- Asserts do not support inheritance.
- Asserts do not yield automatic documentation.

16

## Contracts for analysis, specification



14

## Correctness in software

Correctness is a relative notion: consistency of implementation vis-à-vis 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  $Q$ .

17

## Contracts for analysis

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

15

## 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:

$$P \text{ implies } Q$$

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

18

## 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
```

19

## The contract

Routine	OBLIGATIONS	BENEFITS
Client	PRECONDITION	POSTCONDITION
Supplier	POSTCONDITION	PRECONDITION

22

## 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.  $\{...\} A \{True\}$

20

## A class without contracts

```
class
  ACCOUNT
feature -- Access
  balance: INTEGER
  -- Balance
  Minimum_balance: INTEGER is 1000
  -- Minimum balance
feature {NONE} -- Deposit and withdrawal
  add(sum: INTEGER) is
  -- Add sum to the balance (secret procedure).
  do
    balance := balance + sum
  end
```

23

## 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
```

21

## A class without contracts

```
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
```

24

### 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
  
```

25

### Introducing contracts

```

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
  
```

28

### Introducing contracts

```

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
  
```

26

### The contract

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

29

### Introducing contracts

```

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
  
```

27

### The imperative and the applicative

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

30

### Introducing contracts

```

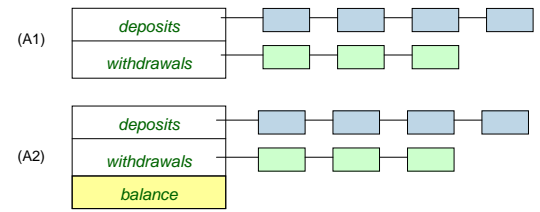
may_withdraw (sum: INTEGER): BOOLEAN is
  -- Is it permitted to withdraw sum from account?
  do
    Result := (balance - sum >= Minimum_balance)
  end

invariant
  not_under_minimum: balance >= Minimum_balance
end

```

31

### Uniform Access



(A1)

(A2)

$balance = deposits.total - withdrawals.total$

34

### 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:  $x.f(\dots)$ )

32

### A slightly more sophisticated version

```

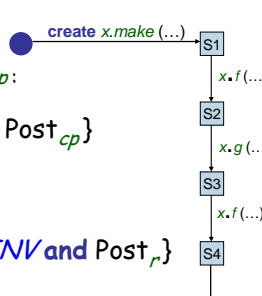
class
  ACCOUNT
create
  make
feature {NONE} -- Implementation
  add (sum: INTEGER) is
    -- Add sum to the balance (secret procedure).
    do
      balance := balance + sum
    ensure
      balance_increased: balance = old balance + sum
    end

  deposits: DEPOSIT_LIST
  withdrawals: WITHDRAWAL_LIST

```

35

### The correctness of a class



For every creation procedure  $cp$ :

$\{Pre_{cp}\} do_{cp} \{INV \text{ and } Post_{cp}\}$

For every exported routine  $r$ :

$\{INV \text{ and } Pre_r\} do_r \{INV \text{ and } Post_r\}$

33

### New version

```

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
      create deposits, make
      create withdrawals, make
    ensure
      balance_set: balance = initial_amount
    end
feature -- Access
  balance: INTEGER
  -- Balance
  Minimum_balance: INTEGER is 1000
  -- Minimum balance

```

36

### New version

```

feature -- Deposit and withdrawal operations

deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  require
    not_too_small: sum >= 0
  do
    add(sum)
    deposits.extend(create {DEPOSIT}.make(sum))
  ensure
    increased: balance = old balance + sum
    one_more: deposits.count = old deposits.count + 1
  end

```

37

### The correctness of a class

```

graph TD
  Start(( )) -- "create x.make(...)" --> S1[S1]
  S1 -- "x.f(...)" --> S2[S2]
  S2 -- "x.g(...)" --> S3[S3]
  S3 -- "x.f(...)" --> S4[S4]
  S4 --> End(( ))

```

For every creation procedure  $cp$ :

$$\{Pre_{cp}\} \text{ do } cp \{INV \text{ and } Post_{cp}\}$$

For every exported routine  $r$ :

$$\{INV \text{ and } Pre_r\} \text{ do } r \{INV \text{ and } Post_r\}$$

40

### New version

```

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)
    withdrawals.extend(create {WITHDRAWAL}.make(sum))
  ensure
    decreased: balance = old balance - sum
    one_more: withdrawals.count = old withdrawals.count + 1
  end

```

38

### Initial version

```

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
    create deposits.make
    create withdrawals.make
  ensure
    balance_set: balance = initial_amount
  end

```

41

### New version

```

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

invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total
end

```

39

### Correct version

```

feature {NONE} -- Initialization

make (initial_amount: INTEGER) is
  -- Set up account with initial_amount.
  require
    large_enough: initial_amount >= Minimum_balance
  do
    create deposits.make
    create withdrawals.make
    deposit (initial_amount)
  ensure
    balance_set: balance = initial_amount
  end

```

42

## What are contracts good for?

Writing correct software (analysis, design, implementation, maintenance, reengineering).  
 Documentation (the "contract" form of a class).  
 Effective reuse.  
 Controlling inheritance.  
 Preserving the work of the best developers.

-----  
 Quality assurance, testing, debugging (especially in connection with the use of libraries).  
 Exception handling.

43

## The contract language

Language of boolean expressions (plus old):

- > No predicate calculus (i.e. no quantifiers,  $\forall$  or  $\exists$ ).
- > Function calls permitted (e.g. in a *STACK* class):

```

put (x: G) is
  -- Push x on top of stack.
  require
    not is_full
  do
    ...
  ensure
    not is_empty
end

remove is
  -- Pop top of stack.
  require
    not is_empty
  do
    ...
  ensure
    not is_full
end
    
```

46

## Contracts: run time effect

Compilation options (per class, in Eiffel):

- > No assertion checking
- > Preconditions only
- > Preconditions and postconditions
- > Preconditions, postconditions, class invariants
- > All assertions

44

## The contract language

First order predicate calculus may be desirable, but not sufficient anyway.

Example: "The graph has no cycles".

In assertions, use only side-effect-free functions.

Use of iterators provides the equivalent of first-order predicate calculus in connection with a library such as EiffelBase or STL. For example (Eiffel agents, i.e. routine objects):

```

my_integer_list.for_all(agent is_positive (?))
with
  is_positive (x: INTEGER): BOOLEAN is
  do
    Result := (x > 0)
  end
    
```

47

## A contract violation is not a special case

For special cases (e.g. "if the sum is negative, report an error...")  
 use standard control structures (e.g. **if ... then ... else...**).

A run-time assertion violation is something else: the manifestation of

**A DEFECT ("BUG")**

45

## The imperative and the applicative

<b>do</b>	<b>ensure</b>
<i>balance := balance - sum</i>	<i>balance = old balance - sum</i>
<b>PRESCRIPTIVE</b>	<b>DESCRIPTIVE</b>
How?	What?
Operational	Denotational
<b>Implementation</b>	<b>Specification</b>
Command	Query
Instruction	Expression
Imperative	Applicative

48

## A contract violation is not a special case

For special cases (e.g. "if the sum is negative, report an error..."), use standard control structures (e.g. **if ... then ... else...**).

A run-time assertion violation is something else: the manifestation of

**A DEFECT ("BUG")**

49

## Contracts and quality assurance

Use run-time assertion monitoring for quality assurance, testing, debugging.

Compilation options (reminder):

- > No assertion checking
- > Preconditions only
- > Preconditions and postconditions
- > Preconditions, postconditions, class invariants
- > All assertions

52

## Contracts and quality assurance

Precondition violation: **Bug in the client.**

Postcondition violation: **Bug in the supplier.**

Invariant violation: **Bug in the supplier.**

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

50

## Contracts missed

Ariane 5 (see Jézéquel & Meyer, IEEE Computer, January 1997)

Lunar Orbiter Vehicle

Failure of air US traffic control system, November 2000

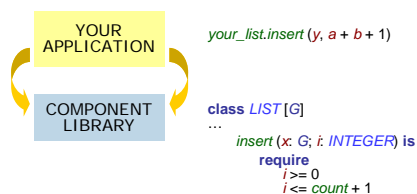
Y2K

etc. etc. etc.

53

## Contracts and bug types

Preconditions are particularly useful to find bugs in **client** code:



51

## Contracts and quality assurance

Contracts enable QA activities to be based on a precise description of what they expect.

Profoundly transform the activities of testing, debugging and maintenance.

*"I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. ... This sort of contract mechanism is the sine-qua-non of sensible software reuse."*

Tom de Marco, IEEE Computer, 1997

54

## Contract monitoring

- Enabled or disabled by compile-time options.
- Default: preconditions only.
- In development: use "all assertions" whenever possible.
- During operation: normally, should disable monitoring. But have an assertion-monitoring version ready for shipping.
- Result of an assertion violation: exception.

Ideally: static checking (proofs) rather than dynamic monitoring.

55

## Class example (continued)

```
feature -- Deposit and withdrawal operations

deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  require
    not_too_small: sum >= 0
  do
    add(sum)
    deposits.extend(create {DEPOSIT}.make(sum))
  ensure
    increased: balance = old balance + sum
  end
```

58

## Contracts and documentation

Recall example class:

```
class
  ACCOUNT
create
  make
feature {NONE} -- Implementation
  add (sum: INTEGER) is
    -- Add sum to the balance (secret procedure).
    do
      balance := balance + sum
    ensure
      increased: balance = old balance + sum
    end
  deposits: DEPOSIT_LIST
  withdrawals: WITHDRAWAL_LIST
```

56

## Class example (continued)

```
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)
    withdrawals.extend(create {WITHDRAWAL}.make(sum))
  ensure
    decreased: balance = old balance - sum
    one_more: withdrawals.count = old withdrawals.count + 1
  end
```

59

## Class example (continued)

```
feature {NONE} -- Initialization
  make (initial_amount: INTEGER) is
    -- Set up account with initial_amount.
    require
      large_enough: initial_amount >= Minimum_balance
    do
      deposit (initial_amount)
      create deposits.make
      create withdrawals.make
    ensure
      balance_set: balance = initial_amount
    end
feature -- Access
  balance: INTEGER
  -- Balance
  Minimum_balance: INTEGER is 1000
  -- Minimum balance
```

57

## Class example (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
  consistent: balance = deposits.total - withdrawals.total
end
```

60

## Contract form: Definition

Simplified form of class text, retaining interface elements only:

- Remove any non-exported (private) feature.

For the exported (public) features:

- Remove body (do clause).
- Keep header comment if present.
- Keep contracts: preconditions, postconditions, class invariant.
- Remove any contract clause that refers to a secret feature. (This raises a problem; can you see it?)

61

## Contract form (continued)

```
withdraw (sum: INTEGER)
  -- Withdraw sum from the account.
  require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Minimum_balance
  ensure
    decreased: balance = old balance - sum
    one_more: withdrawals.count = old withdrawals.count + 1

may_withdraw (sum: INTEGER): BOOLEAN
  -- Is it permitted to withdraw sum from the
  -- account?

invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total

end
```

64

## Export rule for preconditions

In

```
feature {A, B, C}
  r (...) is
    require
      some_property
```

*some\_property* must be exported (at least) to *A*, *B* and *C*.  
No such requirement for postconditions and invariants.

62

## Flat, interface

**Flat form of a class:** reconstructed class with all the features at the same level (immediate and inherited). Takes renaming, redefinition etc. into account.

The flat form is an **inheritance-free client-equivalent form of the class**.

**Interface form:** the contract form of the flat form. Full interface documentation.

65

## Contract form of ACCOUNT class

```
class interface ACCOUNT
  create
    make
  feature
    balance: INTEGER
    -- Balance
    Minimum_balance: INTEGER is 1000
    -- Minimum balance
    deposit (sum: INTEGER)
    -- Deposit sum into the account.
    require
      not_too_small: sum >= 0
    ensure
      increased: balance = old balance + sum
```

63

## Uses of the contract and interface forms

Documentation, manuals  
Design  
Communication between developers  
Communication between developers and managers

66

## Contracts and reuse

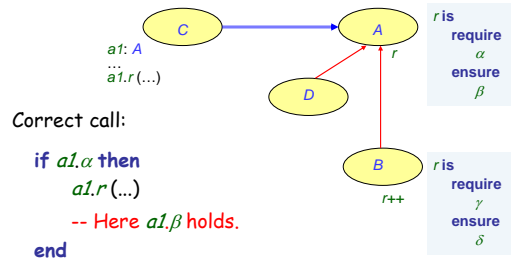
The contract form — i.e. the set of contracts governing a class — should be the standard form of library documentation.

Reuse without a contract is sheer folly.

See the Ariane 5 example.  
See Jézéquel & Meyer, IEEE *Computer*, January 1997.

67

## Contracts and inheritance



70

## Contracts and inheritance

Issues: what happens, under inheritance, to

- > Class invariants?
- > Routine preconditions and postconditions?

68

## Assertion redeclaration rule

When redeclaring a routine:

- > Precondition may only be kept or weakened.
- > Postcondition may only be kept or strengthened.

Redeclaration covers both redefinition and effecting.

Should this remain a purely methodological rule? A compiler can hardly infer e.g. that:

$$n > 1$$

implies (is stronger) than

$$n^{26} + 3 * n^{25} > 3$$

71

## 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.

69

## Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

Redefined version may **not** have **require** or **ensure**.

May have nothing (assertions kept by default), or

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

Resulting assertions are:

- > new\_pre or else original\_precondition
- > original\_postcondition and then new\_post

72

## Don't call us, we'll call you

```

deferred class LIST[E]
inherit
  CHAIN[E]
feature
  has(x: E): BOOLEAN is
    -- Does x appear in list?
    do
      from start
      until after or else found(x)
      loop
        forth
      end
    end
    Result := not after
  end
end

```

73

## Sequential structures (continued)

```

index: INTEGER is
  deferred
  end

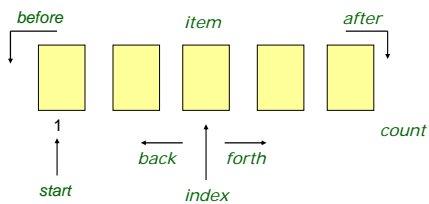
... empty, found, after, ...

invariant
  0 <= index
  index <= size + 1
  empty implies (after or before)
end

```

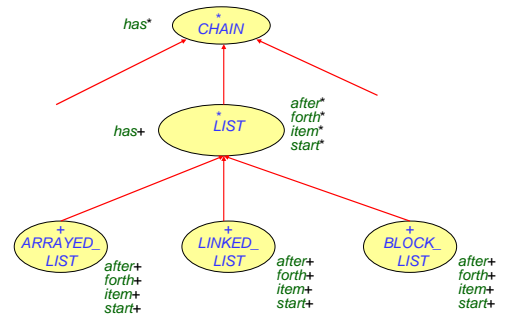
76

## Sequential structures



74

## Descendant implementations



77

## Sequential structures (continued)

```

forth is
  require
    not after
  deferred
  ensure
    index = old index + 1
  end

start is
  deferred
  ensure
    empty or else index = 1
  end
end

```

75

## Implementation variants

	start	forth	after	found (x)
Arrayed list	i := 1	i := i + 1	i > count	t [i] = x
Linked list	c := first_cell	c := c.right	c := Void	c.item = x
File	rewind	read	end_of_file	f↑ = x

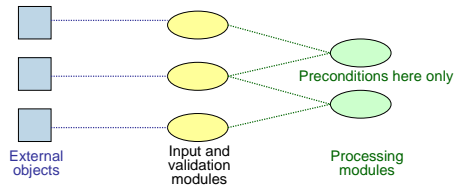
78

## Methodological notes

Contracts are not input checking tests...

... but they can be used to help weed out undesirable input.

Filter modules:



79

## The contract

<i>sqrt</i>	OBLIGATIONS	BENEFITS
<i>Client</i>	(Satisfy precondition:) Provide non-negative value and precision that is not too small.	(From postcondition:) Get square root within requested precision.
<i>Supplier</i>	(Satisfy postcondition:) Produce square root within requested precision.	(From precondition:) Simpler processing thanks to assumptions on value and precision.

82

## Precondition design

The client must **guarantee** the precondition before the call.

This does not necessarily mean **testing** for the precondition.

Scheme 1 (testing):

```
if not my_stack.is_full then
  my_stack.put(some_element)
end
```

Scheme 2 (guaranteeing without testing):

```
my_stack.remove
...
my_stack.put(some_element)
```

80

## Not defensive programming!

It is **never acceptable** to have a routine of the form

```
sqrt(x, epsilon: REAL): REAL is
-- Square root of x, precision epsilon
require
  x >= 0
  epsilon >= 0
do
  if x < 0 then
    ... Do something about it (?) ...
  else
    ... normal square root computation ...
end
ensure
  abs(Result^2 - x) <= 2 * epsilon * Result end
```

83

## Another example

```
sqrt(x, epsilon: REAL): REAL is
-- Square root of x, precision epsilon
require
  x >= 0
do
  ...
ensure
  abs(Result^2 - x) <= 2 * epsilon * Result
end
```

81

## Not defensive programming

For every consistency condition that is required to perform a certain operation:

- > Assign responsibility for the condition to one of the contract's two parties (supplier, client).
- > Stick to this decision: do not duplicate responsibility.

Simplifies software and improves global reliability.

84

### Interpreters

```

class BYTECODE_PROGRAM
feature
  verified: BOOLEAN
  trustful_execute(program: BYTECODE) is
  require
    do ok: verified
    ...
  end
  distrustful_execute(program: BYTECODE) is
  do
    verify
    if verified then
      trustful_execute(program)
    end
  end
  verify is
  do
    ...
  end
end
end

```

85

### A tolerant style

```

sqrt(x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
  True
  if x < 0 then
    ... Do something about it (?) ...
  else
    ... normal square root computation ...
    computed := True
  end
  ensure
  computed implies
    abs(Result^2 - x) <= 2 * epsilon * Result
  end

```

NO INPUT TOO BIG OR TOO SMALL!

88

### How strong should a precondition be?

Two opposite styles:

- > **Tolerant:** weak preconditions (including the weakest, *True*: no precondition).
- > **Demanding:** strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call.

Partly a matter of taste.

But: demanding style leads to a better distribution of roles, provided the precondition is:

- > Justifiable in terms of the specification only.
- > Documented (through the short form).
- > Reasonable!

86

### Contrasting styles

```

put(x: E) is
  -- Push x on top of stack.
  require
  not is_full
  do
    ...
  end

tolerant_put(x: E) is
  -- Push x if possible, otherwise set impossible to
  -- True.
  do
    if not is_full then
      put(x)
    else
      impossible := True
    end
  end

```

89

### A demanding style

```

sqrt(x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  -- Same version as before
  require
  x >= 0
  do
    ...
  end
  ensure
  abs(Result^2 - x) <= 2 * epsilon * Result
  end

```

epsilon

87

### Invariants and "business rules"

Invariants are absolute consistency conditions.

They can serve to represent business rules if knowledge is to be built into the software.

Form 1

```

invariant
  not_under_minimum: balance >= Minimum_balance

```

Form 2

```

invariant
  not_under_minimum_if_normal:
    normal_state implies
      (balance >= Minimum_balance)

```

90

## A powerful assertion language

Assertion language:

- Not first-order predicate calculus
- But powerful through:
  - Function calls
- Even allows to express:
  - Loop properties

91

## Design by Contract: technical benefits

Development process becomes more focused. Writing to spec.  
Sound basis for writing reusable software.  
Exception handling guided by precise definition of "normal" and "abnormal" cases.  
Interface documentation always up-to-date, can be trusted.  
Documentation generated automatically.  
Faults occur close to their cause. Found faster and more easily.  
Guide for black-box test case generation.

94

## Another contract mechanism

**Check instruction:** ensure that a property is True at a certain point of the routine execution.

E.g. Tolerant style example: Adding a check clause for readability.

92

## Managerial benefits

Library users can trust documentation.  
They can benefit from preconditions to validate their own software.  
Test manager can benefit from more accurate estimate of test effort.  
Black-box specification for free.  
Designers who leave bequeath not only code but intent.  
Common vocabulary between all actors of the process: developers, managers, potentially customers.  
Component-based development possible on a solid basis.

95

## Precondition design

Scheme 2 (guaranteeing without testing):

```
my_stack.remove
check
  my_stack_not_full: not my_stack.is_full
end
my_stack.put(some_element)
```

93

## Exception handling

**The need for exceptions arises when the contract is broken.**

Two concepts:

- **Failure:** a routine, or other operation, is unable to fulfill its contract.
- **Exception:** an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.

96

## Analysis classes

```
deferred class
  VAT
inherit
  TANK
feature
  in_valve, out_valve: VALVE
  fill is
    require
      -- Fill the vat.
      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
```

97

## What Requirements analysis is not

Use cases

(Not appropriate as requirements statement mechanism)

Use cases are to requirements what tests are to specification and design

10

## What is object oriented analysis?

- **Classes** around object types (not just physical objects but also important concepts of the application domain)
- **Abstract Data Types** approach
- **Deferred** classes and features
- Inter-component relations: "**client**" and inheritance
- Distinction between **reference** and **expanded** clients
- **Inheritance** — single, multiple and repeated for classification.
- **Contracts** to capture the *semantics* of systems: properties other than structural.
- **Libraries** of reusable classes

98

## Television station example

Source: OOSC

```
class SCHEDULE feature
  segments: LIST [SEGMENT]
end
```

10

## Why Analysis?

Same benefits as O-O programming, in particular extendibility and reusability

Direct modeling of the problem domain

Seamlessness and reversibility with the continuation of the project (design, implementation, maintenance)

99

## Schedules

```
note
  description:
    "24-hour TV schedules"
deferred class SCHEDULE feature
  segments: LIST [SEGMENT]
    -- Successive segments
  deferred
  end
  air_time: DATE is
    -- 24-hour period
    -- for this schedule
  deferred
  end
  set_air_time (t: DATE)
    -- Assign schedule to
    -- be broadcast at time t.
  require
    t.in_future
  deferred
  ensure
    air_time = t
  end
  print
    -- Produce paper version.
  deferred
  end
end
```

10

## Contracts

Feature precondition: condition imposed on the rest of the world

Feature postcondition: condition guaranteed to the rest of the world

Class invariant: Consistency constraint maintained throughout on all instances of the class

10

## Segment

```

note
  description:
    "Individual fragments of a schedule"
deferred class SEGMENT feature
  schedule: SCHEDULE deferred end
  -- Schedule to which
  -- segment belongs
  index: INTEGER deferred end
  -- Position of segment in
  -- its schedule
  starting_time, ending_time:
    INTEGER is deferred end
  -- Beginning and end of
  -- scheduled air time
  next: SEGMENT is deferred end
  -- Segment to be played
  -- next, if any

  sponsor: COMPANY deferred end
  -- Segment's principal sponsor

  rating: INTEGER deferred end
  -- Segment's rating (for
  -- children's viewing etc.)

  ... Commands such as change_next,
  set_sponsor, set_rating omitted ...

  Minimum_duration: INTEGER = 30
  -- Minimum length of segments,
  -- in seconds

  Maximum_interval: INTEGER = 2
  -- Maximum time between two
  -- successive segments, in seconds
  
```

10

## Obligations & benefits in a contract

<i>deliver</i>	OBLIGATIONS	BENEFITS
<i>Client</i>	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
<i>Supplier</i>	(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.

10

## Segment (continued)

```

invariant
  in_list: (1 <= index) and (index <= schedule.segments.count)
  in_schedule: schedule.segments.item(index) = Current
  next_in_list: (next /= Void) implies
    (schedule.segments.item(index + 1) = next)

  no_next_iff_last: (next = Void) = (index = schedule.segments.count)
  non_negative_rating: rating >= 0
  positive_times: (starting_time > 0) and (ending_time > 0)
  sufficient_duration:
    ending_time - starting_time >= Minimum_duration
  decent_interval:
    (next.starting_time) - ending_time <= Maximum_interval
end
  
```

10

## Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:

- > Committing too early to an implementation  
Overspecification!
- > Missing parts of the problem  
Underspecification!

)

10

## Commercial

```

note
  description: "Advertising segment"
deferred class COMMERCIAL inherit
  SEGMENT
  rename sponsor as advertiser end
feature
  primary: PROGRAM deferred
  -- Program to which this
  -- commercial is attached
  primary_index: INTEGER deferred
  -- Index of primary

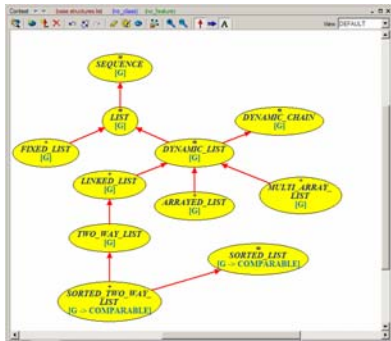
  set_primary(p: PROGRAM)
  -- Attach commercial to p.
  require
    program_exists: p /= Void
    same_schedule: p.schedule = schedule
  before:
    p.starting_time <= starting_time
  deferred
  ensure
    index_updated:
      primary_index = p.index
    primary_updated: primary = p
  end

invariant
  meaningful_primary_index: primary_index = primary.index
  primary_before: primary.starting_time <= starting_time
  acceptable_sponsor: advertiser.compatible(primary.sponsor)
  acceptable_rating: rating <= primary.rating
end
  
```

10

## Diagrams: UML, BON

### Text-Graphics Equivalence



10

## Causes of exceptions

Assertion violation

Void call ( $x.f$  with no object attached to  $x$ )

Operating system signal (arithmetic overflow, no more memory, interrupt ...)

11

## Analysis process

Identify abstractions

- > New
- > Reused

Describe abstractions through interfaces, with contracts

Look for more specific cases: use inheritance

Look for more general cases: use inheritance, simplify

Iterate on suppliers

At all stages keep structure simple and look for applicable contracts

11

## Handling exceptions properly

Safe exception handling principle:

- > There are only two acceptable ways to react for the recipient of an exception:
  - Concede failure, and trigger an exception in the caller (**Organized Panic**).
  - Try again, using a different strategy (or repeating the same strategy) (**Retrying**).

11

## The original strategy

```
r(...) is
  require
  ...
  do
    op1
    op2
    ...
    op
    ...
    opn
  ensure
  ...
end
```

← Fails, triggering an exception in  $r$  ( $r$  is recipient of exception).

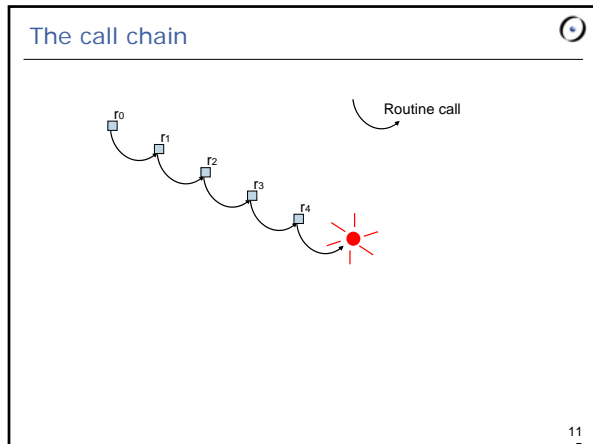
11

## How not to do it

(From an Ada textbook)

```
sqrt(x: REAL) return REAL is
begin
  if x < 0.0 then
    raise Negative;
  else
    normal_square_root_computation;
  end
exception
  when Negative =>
    put("Negative argument");
    return;
  when others => ...
end; -- sqrt
```

11



### Transmitting over an unreliable line (2)

```

Max_attempts: INTEGER is 100
failed: BOOLEAN

attempt_transmission(message: STRING) is
  -- Try to transmit message.
  -- if impossible in at most Max_attempts
  -- attempts, set failed to true.

  local failures: INTEGER
  do
    if failures < Max_attempts then
      unsafe_transmit(message)
    else
      failed := True
    end
  rescue
    failures := failures + 1
  retry
end

```

11

### Exception mechanism

Two constructs:

- > A routine may contain a **rescue** clause.
- > A rescue clause may contain a **retry** instruction.

A **rescue** clause that does not execute a **retry** leads to failure of the routine (this is the organized panic case).

11

### If no exception clause (1)

Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

```

f(...) is
  do
    ...
  end

```

is an abbreviation for

```

f(...) is
  do
    ...
  rescue
    -- Nothing here
  end

```

(This is a provisional rule; see next.)

11

### Transmitting over an unreliable line (1)

```

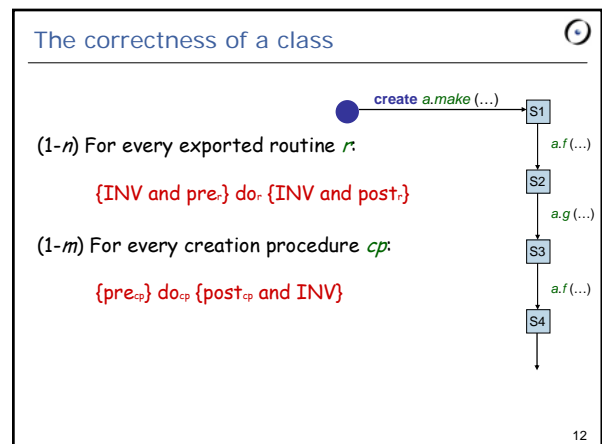
Max_attempts: INTEGER is 100

attempt_transmission(message: STRING) is
  -- Transmit message in at most
  -- Max_attempts attempts.

  local failures: INTEGER
  do
    unsafe_transmit(message)
  rescue
    failures := failures + 1
    if failures < Max_attempts then
      retry
    end
  end
end

```

11



### Exception correctness: A quiz

For the normal body:

```
{INV and pre.} do. {INV and post.}
```

For the exception clause:

```
{ ??? } rescue. { ??? }
```

12

### If no exception clause (2)

Absence of a rescue clause is equivalent to a default rescue clause:

```
f(...) is
do
...
end
```

is an abbreviation for

```
f(...) is
do
...
rescue
default_rescue
end
```

The task of *default\_rescue* is to restore the invariant.

12

### Quiz answers

For the normal body:

```
{INV and pre.} do. {INV and post.}
```

For the exception clause:

```
{ True } rescue. { INV }
```

12

### For finer gain exception handling

Use class *EXCEPTIONS* from the Kernel Library.

Some features:

- > *exception* (code of last exception that was triggered).
- > *is\_assertion\_violation*, etc.
- > *raise* ("exception\_name")

12

### Bank accounts

$balance := deposits.total - withdrawals.total$

(A2)

12

### Agenda for today

Exception handling

Design by Contract outside of Eiffel

12

## Design by Contract outside of Eiffel

Basic step: use standardized comments, or graphical annotations, corresponding to **require**, **ensure**, **invariant** clauses.

In programming languages:

- > **Macros**
- > **Preprocessor**

Use of macros avoids the trouble of preprocessors, but invariants are more difficult to handle than preconditions and postconditions.

Difficulties: contract inheritance; "short"-like tools; link with exception mechanism.

12

## Nana example

```
void gsort(int v[], int n) { /* sort v[0..n-1] */
  DI(v != NULL && n >= 0); /* check arguments under gdb(1) only */
  L("gsort(%p, %d)\n", v, n); /* log messages to a circular buffer */
  ... /* the sorting code */
  I(A(int i = 1, i < n, i++, /* verify v[] sorted (forall) */
    v[i-1] <= v[i])); /* forall i in 1..n-1 @ v[i-1] <= v[i] */
}

void intsqr(int &r) { /* r' = floor(sqrt(r)) */
  DS($r = r); /* save r away into $r for later use under gdb(1) */
  DS($start = $cycles); /* real time constraints */
  ... /* code which changes r */
  DI($cycles - $start < 1000); /* code must take less than 1000 cycles */
  DI((r * r) <= $r && ($r < (r + 1) * (r + 1))); /* use $r in postcondition */
}
```

13

## C++/Java Design by Contract limitations

The possibility of direct assignments

```
x.attrib = value
```

limits the effectiveness of contracts: circumvents the official class interface of the class. In a fully O-O language, use:

```
x.set_attrib (value)
```

with

```
set_attrib (v: TYPE) is
  -- Make v the next value for attrib.
  require
  ... Some condition on v ...
  do
    attrib := v
  ensure
    attrib = v
  end
```

12

## Nana

In the basic package: no real notion of class invariant. ("Invariant", macro DI, is equivalent of "check" instruction.)

Package eiffel.h "is intended to provide a similar setup to Eiffel in the C++ language. It is a pretty poor emulation, but it is hopefully better than nothing."

Macros: CHECK\_NO, CHECK\_REQUIRE, CHECK\_ENSURE, CHECK\_INVARIANT, CHECK\_LOOP, CHECK\_ALL.

Using CHECK\_INVARIANT assumes a boolean-valued class method called invariant. Called only if a REQUIRE or ENSURE clause is present in the method.

No support for contract inheritance.

13

## C++ Contracts

GNU Nana: improved support for contracts and logging in C and C++.

Support for quantifiers (forall, Exists, Exists1) corresponding to iterations on the STL (C++ Standard Template Library).

Support for time-related contracts ("Function must execute in less than 1000 cycles").

12

## Java

OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to "lack of time".

"Assert" instruction recently added.

Gosling (May 1999):

- > "The number one thing people have been asking for is an assertion mechanism. Of course, that [request] is all over the map: There are people who just want a compile-time switch. There are people who ... want something that's more analyzable. Then there are people who want a full-blown Eiffel kind of thing. We're probably going to start up a study group on the Java platform community process."

(<http://www.javaworld.com/javaworld/javaone99/1-99-gosling.html>)

13

## iContract

Reference: Reto Kramer. "iContract, the Java Design by Contract Tool". *TOOLS USA 1998*, IEEE Computer Press, pages 295-307.

Java preprocessor. Assertions are embedded in special comment tags, so iContract code remains valid Java code in case the preprocessor is not available.

Support for Object Constraint Language mechanisms.

Support for assertion inheritance.

13

## Another Java tool: Jass (Java)

Preprocessor. Also adds Eiffel-like exception handling.

<http://theoretica.Informatik.Uni-Oldenburg.DE/~jass>

```
public boolean contains(Object o) {
    /** require o != null; */
    for (int i = 0; i < buffer.length; i++)
        /** invariant 0 <= i && i <= buffer.length; */
        /** variant buffer.length - i */
        if (buffer[i].equals(o)) return true;
    return false;
    /** ensure changeonly(); */
}
```

13

## iContract example

```
/**
 * @invariant age_ > 0
 */
class Person {
    protected age_;
    /**
     * @post return > 0
     */
    int getAge() {...}
    /**
     * @pre age > 0
     */
    void setAge( int age ){...}
    ...
}
```

13

## Biscotti

Adds assertions to Java, through modifications of the JDK 1.2 compiler.

Cynthia della Torre Cicalese

See *IEEE Computer*, July 1999

13

## iContract specification language

Any expression that may appear in an `if(...)` condition may appear in a precondition, postcondition or invariant.

### Scope:

- Invariant: as if it were a routine of the class.
- Precondition and postcondition: as if they were part of the routine.

### OCL\*-like assertion elements

- forall Type t in <enumeration> | <expr>
- exists Type t in <enumeration> | <expr>
- <a> implies <b>

(\* OCL: Object Constraint Language)

13

## The Object Constraint Language

Designed by IBM and other companies as an addition to UML.

Includes support for:

- Invariants, preconditions, postconditions
- Guards (not further specified).
- Predefined types and collection types
- Associations
- Collection operations: ForAll, Exists, Iterate

Not directly intended for execution.

Jos Warmer, AW

13

## OCL examples

### Postconditions:

```
post: result = collection->iterate  
(elem; acc : Integer = 0 | acc + 1)
```

```
post: result = collection->iterate  
( elem; acc : Integer = 0 |  
  if elem = object then acc + 1 else acc endif)
```

```
post: T.allInstances->forAll  
(elem | result->includes(elem) = set->  
  includes(elem) and set2->includes(elem))
```

Collection types include Collection, Set, Bag, Sequence.

13

## Contracts for COM and Corba

See: Damien Watkins. "Using Interface Definition Languages to support Path Expressions and Programming by Contract". *TOOLS USA 1998*, IEEE Computer Press, pages 308-317.

Set of mechanisms added to IDL to include: preconditions, postconditions, class invariants.

14

## Complementary material

### OOSC2:

- [Chapter 11: Design by Contract](#)

14