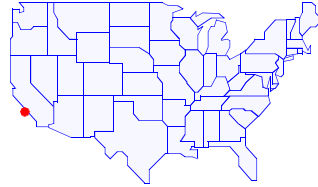


DESIGN BY CONTRACT and Genericity for Java

Bertrand Meyer
Interactive Software Engineering & Monash University

Author's address:

Interactive Software Engineering
ISE Building, 270 Storke Road
Santa Barbara, CA 93117 USA
Telephone 805-685-1006, Fax 805-685-6869
E-mail <info@eiffel.com> <http://tools.com>



<http://eiffel.com>

© Bertrand Meyer, 1988-1999

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

1

PLAN

- Part 1: Introduction and overview 3
- Part 2: Design by Contract 8
- Part 3: Genericity 87

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

2

PART 1:

INTRODUCTION
AND OVERVIEW

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

3

JAVA LANGUAGE PHILOSOPHY

- C++-like in appearance
- Grew beyond original goals
- Simpler than C++
- Supports garbage collection
- Neither small nor large: able to grow

Guy Steele:

"I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow but I need, too, to leave some choices so that other persons can make those choices at a later time."

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

4

WHO IS IN CHARGE?

Sun

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

5

JAVA CRITICISM AND REQUESTS

True multiple inheritance

Design by Contract

Genericity

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

6

THE JAVA REQUEST PAGE



CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

7

PART 2:

DESIGN
by
CONTRACT

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

8

DESIGN BY CONTRACT

A systematic method for making software reliable.

Applications:

- Better analysis and design.
- Implementation.
- Testing, debugging, quality assurance.
- Documentation.
- Basis for exception handling.
- Controlling inheritance (sub-contracting).
- Project management: preserving top designers' work.
- Built-in reliability.
(Harlan Mills, 1975: "*How to write correct programs, and know it*".)

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

9

DESIGN BY CONTRACT: SCOPE

Methodological principles are language- and tool-independent.

Applications to debugging, quality assurance, testing, documentation, exception handling, inheritance require language and tool support:

- Built-in in the Eiffel language, the BON analysis & design method & notation, and the supporting tools (EiffelBench, EiffelCase). See also Catalysis and OCL.
- Can be partially emulated in C++ through macros.
- Various proposed extensions for Java.
- Extensions proposed for other languages.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

10

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.

(For a counter-example see the Ariane case).

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

11

A NEW VIEW OF SOFTWARE CONSTRUCTION

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

These definitions are the contracts.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

12

PROPERTIES OF CONTRACTS

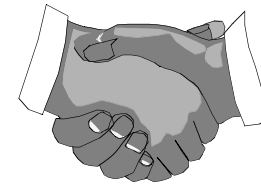
A contract:

- Binds two parties (or more): client, supplier.
- 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).

CONT 99-10

13

A HUMAN CONTRACT



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

CONT 99-10

14

AN ANALYSIS CONTRACT

```

deferred class PLANE inherit
  AIRCRAFT
feature
  start_take_off is
    -- Initiate take-off procedures.
    require
      controls.passed; assigned_runway.clear
    deferred
    ensure
      assigned_runway.owner = Current
      moving
    end
  start_landing, increase_altitude, decrease_altitude,
  moving, altitude, speed, time_since_take_off
  ... [Other features] ...
invariant
  (time_since_take_off <= 120) implies (assigned_runway.owner = Current)
  moving = (speed > 10)
end
    
```

Annotations:

- Precondition**: points to `controls.passed; assigned_runway.clear`
- Postcondition**: points to `assigned_runway.owner = Current`
- Class invariant**: points to `(time_since_take_off <= 120) implies (assigned_runway.owner = Current)`
- Deferred**: points to `deferred` (i.e. specified only, not implemented)

CONT 99-10

15

AN ANALYSIS CONTRACT

```

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
    
```

Annotations:

- Precondition**: points to `in_valve.open; out_valve.closed`
- Postcondition**: points to `in_valve.closed; out_valve.closed; is_full`
- Class invariant**: points to `is_full = ((gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum))`
- Deferred**: points to `deferred` (i.e. specified only, not implemented)

CONT 99-10

16

RUNWAYSdeferred class **RUNWAY** feature**owner: AIRCRAFT****clear: BOOLEAN**... **[Other features]** ...

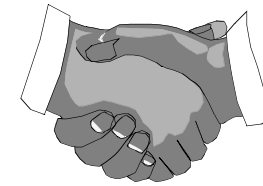
invariant

clear_iff_owned: not **clear** = (**owner** /= Void)

end

CONT 99-10

17

CONTRACTS FOR ANALYSIS

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

CONT 99-10

18

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.

CONT 99-10

19

**DESIGN BY CONTRACT: BENEFITS
1: TECHNICAL**

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.

CONT 99-10

20

DESIGN BY CONTRACT: BENEFITS 2: MANAGERIAL

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.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

21

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)?

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

22

THE FRENCH DRIVER'S LICENSE SYNDROME

1. Nom: MEYER	CATEGORIES DE VEHICULES SUR LESQUELS LE Titulaire I.D. a
2. Prénoms: BERTRAND	
3. Date et lieu de naissance: 21/11/50 PARIS	
4. Domicile: PARIS CITE D'HAUTEVILLE PARIS 10	
Signature de l'usager  Délivré par: Préfet de Police Paris Le: 28/9/70 N°: 75/1 929 474 LE DIRECTEUR	
	
Véhicul Motoscycles avec ou sans side-car Véhicules de moins de 10 places au d'un poids total en charge n'excédant pas 3.500 kgs Véhicules à marchandises de plus de 3.500 kgs Véhicules de transport en commun (plus de 9 places) Véhicules des catégories R.C.D.F. attelle d'une remorque de plus de 750	

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

23

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`

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

24

A CLASS WITH CONTRACTS

```

class ACCOUNT create
  make
feature -- Initialization
  make (initial_amount: INTEGER) is
    -- Set up account with initial_amount
    require
      large_enough: initial_amount >= Minimum_balance
    do
      balance := initial_amount
    ensure
      set: balance = initial_amount
    end
end

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

25

INTRODUCING CONTRACTS (CONTINUED)

```

feature -- Balance and minimum balance
  balance: INTEGER
  Minimum_balance: INTEGER is 1000
feature {NONE} -- Implementation of deposit and withdrawal

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

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

26

WITH CONTRACTS (CONTINUED)

```

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

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

27

WITH CONTRACTS (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)      -- i.e. balance := balance - sum
    ensure
      decreased: balance = old balance - sum
    end

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

28

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 that sum is within allowable bounds

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

29

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

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

30

WITH CONTRACTS (END)

may_withdraw (sum: INTEGER): BOOLEAN is

```
-- Is it possible to withdraw sum from the account?
do
  Result := (balance >= Minimum_balance + sum)
end
```

invariant

```
not_under_minimum: balance >= Minimum_balance
```

end -- class ACCOUNT

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

31

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.

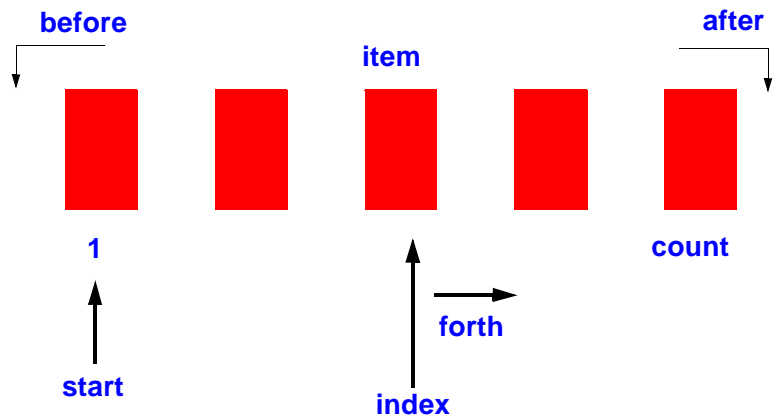
(Qualified calls only: **a.f (...)**)

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

32

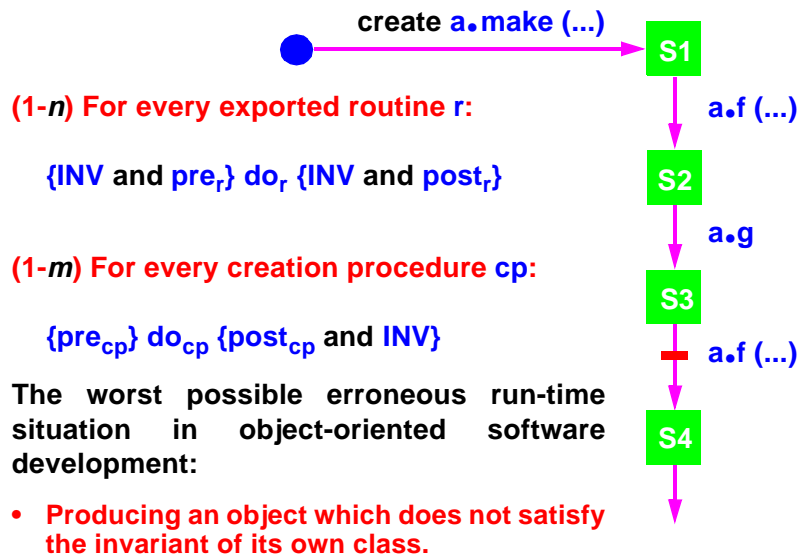
LIST STRUCTURES



CONT 99-10

33

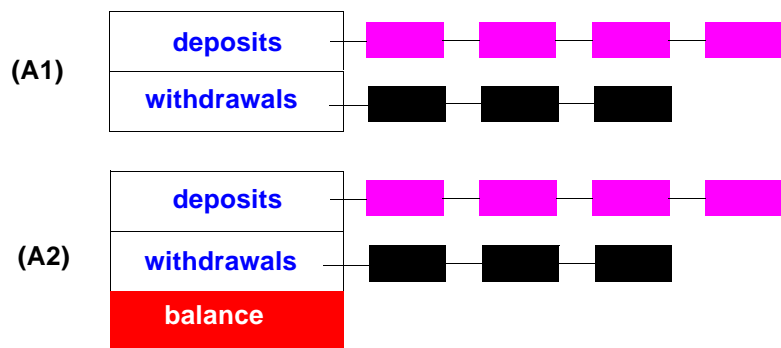
THE CORRECTNESS OF A CLASS



CONT 99-10

34

UNIFORM ACCESS: AN EXAMPLE — BANK ACCOUNTS



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

CONT 99-10

35

A MORE SOPHISTICATED VERSION

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

CONT 99-10

36

NEW VERSION (CONTINUED)

```

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
    end

feature -- Balance and minimum balance
  balance: INTEGER
  Minimum_balance: INTEGER is 1000

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

37

NEW VERSION (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 ({DEPOSIT} .make (sum))

  ensure
    increased: balance = old balance + sum
  end

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

38

NEW VERSION (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 ({WITHDRAWAL} .make (sum))

  ensure
    decreased: balance = old balance - sum
  end

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

39

NEW VERSION (END)

```

may_withdraw (sum: INTEGER): BOOLEAN is
  -- Is it possible to withdraw sum from the account?
  do
    Result := (balance >= Minimum_balance + sum)
  end
invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total

end -- class ACCOUNT

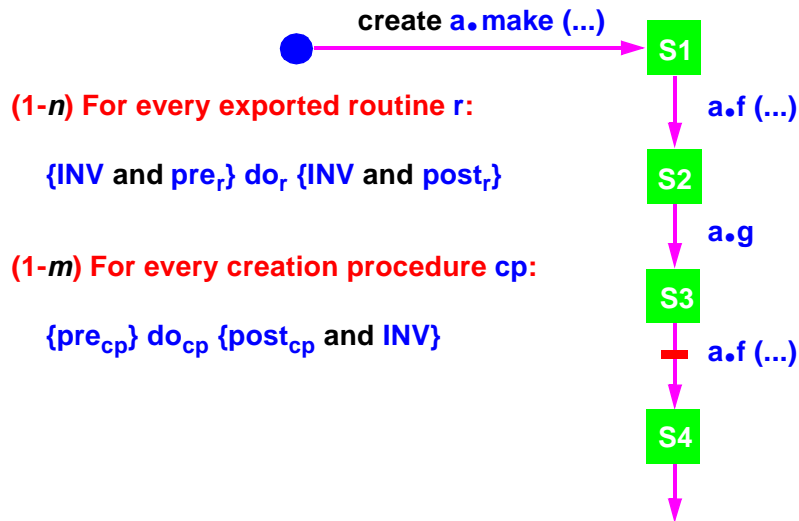
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

40

THE CORRECTNESS OF A CLASS

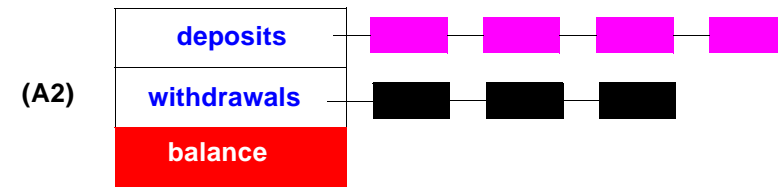


CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

41

SECOND BANK ACCOUNT REPRESENTATION



balance = deposits.total – withdrawals.total

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

42

WHAT ARE CONTRACTS GOOD FOR?

- Writing correct software (analysis, design, implementation, maintenance, reengineering).
 - Documentation (the “short” 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

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

43

CONTRACTS: RUN-TIME EFFECT

Compilation options (per class, in Eiffel):

- 1• No assertion checking
- 2• Preconditions only
- 3• Preconditions and postconditions
- 4• Preconditions, postconditions, class invariants
- 5• All assertions

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

44

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”)

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

45

CONTRACTS AND QUALITY ASSURANCE

Precondition violation: **BUG IN THE CLIENT.**

Postcondition violation: **BUG IN THE SUPPLIER.**

Invariant violation: **BUG IN THE SUPPLIER.**

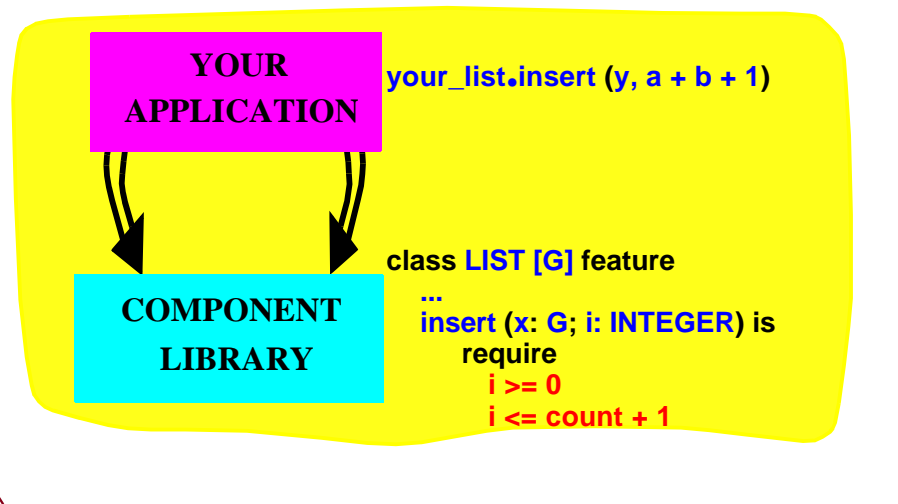
CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

46

CONTRACTS AND BUG TYPES

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



CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

47

CONTRACTS AND QUALITY ASSURANCE

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

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

Profoundly transform the activities of testing, debugging and maintenance.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

48

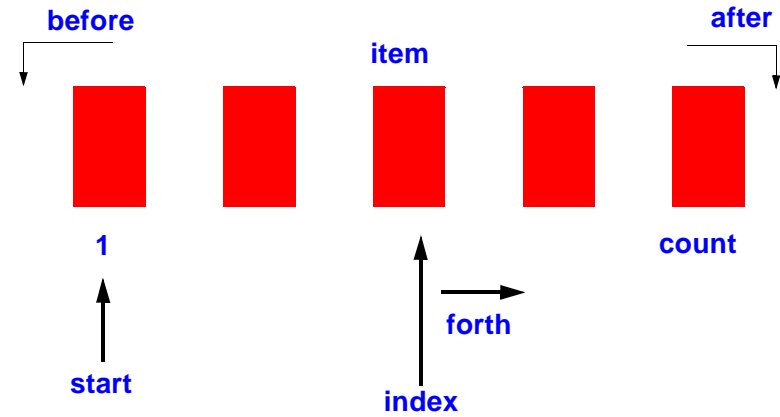
DEBUGGING WITH CONTRACTS: AN EXAMPLE

This example will use a live demo from ISE's EiffelBench, with a "planted" error leading to a precondition violation.

The example uses both the browsing and debugging mechanisms.

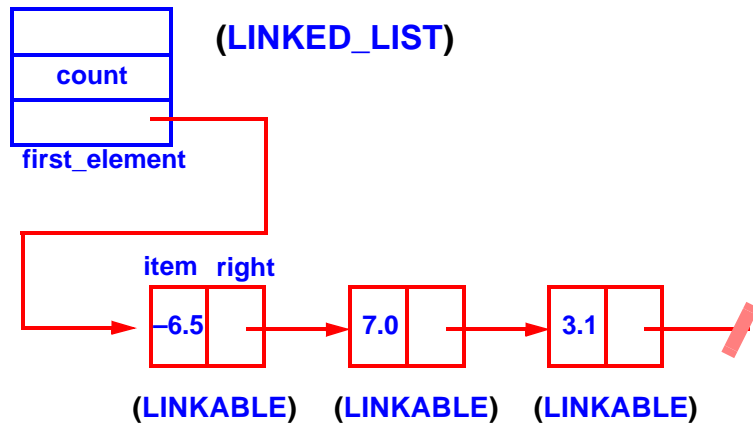
CONT 99-10

49

**TO UNDERSTAND THE EXAMPLE:
LIST CONVENTIONS**

CONT 99-10

50

LINKED LIST REPRESENTATION

CONT 99-10

51

ADDING AND CATCHING A BUG

In class **STARTER**, procedure `make_a_list`, replace the first call to `extend` by a call to `put`.

Execute system. What happens?

Use browsing mechanisms to find out what's wrong (violated precondition).

To understand, consider what the diagram of page 50 becomes when the number of list items goes to zero.

CONT 99-10

52

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.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

53

PART 2.1: 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
        end

    deposits: DEPOSIT_LIST
    withdrawals: WITHDRAWAL_LIST
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

54

CLASS EXAMPLE (CONTINUED)

feature -- 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
    end
```

feature -- Balance and minimum balance

```
balance: INTEGER
Minimum_balance: INTEGER is 1000
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

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 ({DEPOSIT}.make (sum))
    ensure
        increased: balance = old balance + sum
    end
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

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({WITHDRAWAL}.make (sum))
  ensure
    decreased: balance = old balance - sum
  end

```

CONT 99-10

57

CLASS EXAMPLE (END)

```

may_withdraw (sum: INTEGER): BOOLEAN is
  -- Is it possible to withdraw sum from the account?
  do
    Result := (balance >= Minimum_balance + sum)
  end
invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total
end -- class ACCOUNT

```

CONT 99-10

58

SHORT FORM: DEFINITION

See also: JavaDoc

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?)

CONT 99-10

59

EXPORT RULE FOR PRECONDITIONS

In

```

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

```

some_property must be exported (at least) to **A, B** and **C!**

No such requirement for postconditions and invariants.

CONT 99-10

60

SHORT FORM OF ACCOUNT CLASS

```

class interface ACCOUNT create
  make (initial_amount: INTEGER)
    -- Set up account with initial_amount.
  require
    initial_amount >= 0

  feature

    balance: INTEGER

    Minimum_balance: INTEGER is 1000

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

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

61

SHORT FORM (CONTINUED)

```

  withdraw (sum: INTEGER)
    -- Withdraw sum from account.
  require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Minimum_balance
  ensure
    decreased: balance = old balance - sum

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

  invariant

    not_under_minimum: balance >= Minimum_balance

end -- class interface ACCOUNT
  
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

62

FLAT, FLAT-SHORT

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.

Flat-short form: the short form of the flat form. Full interface documentation.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

63

USES OF THE (FLAT-)SHORT FORM

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

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

64

CONTRACTS AND REUSE

The short 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.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

65

CONTRACTS AND INHERITANCE

THE INVARIANT RULE

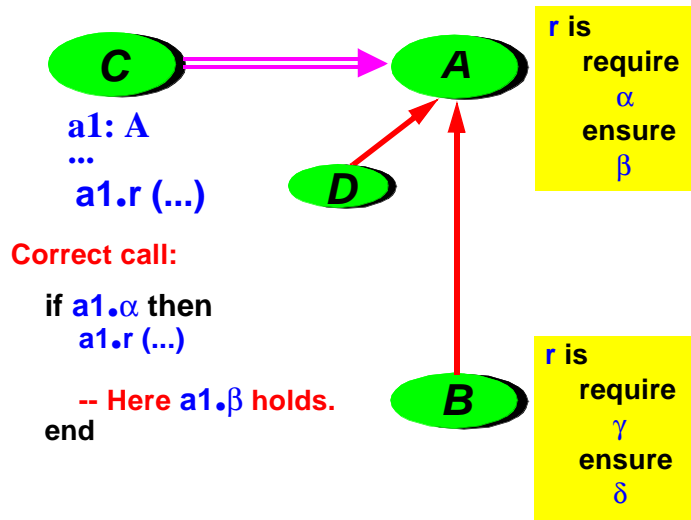
The invariant of a class automatically includes the invariant clauses from all its parents, “and”-ed.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

66

CONTRACTS AND INHERITANCE



CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

67

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.

$n > 0$

$n^{26} + 3 * n^{25} > 4$

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

68

ASSERTION REDECLARATION RULE (EIFFEL)

Redeclared 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:

original_precondition or **new_pre**

original_postcondition and **new_post**

CONT 99-10

69

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.

CONT 99-10

70

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).

CONT 99-10

71

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
```

CONT 99-10

72

AN EXAMPLE PROJECT

Laser printer software at Hewlett-Packard

1997-1998

Embedded system development: software runs on chip in printer

Host development environment: VxWorks operating system

About 800,000 lines of legacy C code.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

73

INTRODUCING DESIGN BY CONTRACT

First in C and C++ through macros.

Eiffel introduced later, in particular because of memory management requirements. C calls Eiffel (CECIL library).

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

74

BENEFITS

Greatly decreased error rates in the elements built with Design by Contract.

Several major errors found in the legacy C code.

Bug in chip.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

75

DESIGN BY CONTRACT IN JAVA

- OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.
- “No assertions” recently moved to #2 on the official Java users’ bug and enhancement request list

“Evaluation: We are currently looking at this to determine if it's appropriate for a future release”.

(<http://developer.java.sun.com/developer/bugParade/bugs/4071460.html>)

- Several different proposals. 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/j1-99-gosling.html>)

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

76

iContract

Reference: **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.

CONT 99-10

77

iContract example

(See attached paper.)

```

/**                               int getAge() ..}
 * @invariant age_ > 0           /**
 */                               * @pre age > 0
                                   */

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

```

CONT 99-10

78

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

*Object Constraint Language

forall Type t in <enumeration> | <expr>

exists Type t in <enumeration> | <expr>

<a> implies

CONT 99-10

79

ANOTHER JAVA TOOL: JASS (JAWA)

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{}; */
}

```

CONT 99-10

80

MORE JAWA/JASS EXAMPLES

```

class Alpha {
  int x,y,z
  ...
  private void Div()
  throws RuntimeException.AssertionException {
    /* check z!=0 */
    x=y/z;
    /* rescue catch (RuntimeException e) {
      z!=1; retry
    }
  }
  */
}
...
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

81

JASS BUFFER

```

package jass.examples;
public class Buffer implements Cloneable {
  protected int in,out;
  protected Object[] buffer;
  public Buffer() {
    buffer = new Object[0];
  }
  public Buffer (int count) {
    /* require count > 0; */
    buffer = new Object[count];
    /* ensure buffer.length == count; */
  }
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

82

JASS BUFFER (CONTINUED)

```

public boolean empty() {
  return in - out == 0;
  /* ensure changeonly{}; */
}
public boolean full() {
  return in - out == buffer.length;
  /* ensure changeonly{}; */
}
public void add(Object o) {
  /* require [valid_object] o != null; [buffer_not_full] !full(); */
  buffer[in % buffer.length] = o;
  in++;
  /* ensure changeonly{in,buffer}; Old.in == in - 1; */
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

83

JASS BUFFER (CONTINUED)

```

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

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

84

JASS BUFFER (END)

```
protected Object clone() {
    /* Use the Objects clone method. This is enough because
       only in and out are referenced in Old expressions. */
    Object b = null;
    try {
        b = super.clone();
    }
    catch (CloneNotSupportedException e){;}
    return b;
}

/** invariant [range] 0 <= in - out && in - out <= buffer.length;
    [valid_content] buffer.length == 0 ||
    (forall i : {out%buffer.length .. in%buffer.length-1}
     # buffer[i] != null); */
```

CONT 99-10

85

BISCOTTI

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

Cynthia della Torre Cicalese

See IEEE *Computer*, July 1999

CONT 99-10

86

PART 3:**GENERICITY**

CONT 99-10

87

GENERICITY:**ENSURING TYPE SAFETY FOR CONTAINER STRUCTURES**

How can we define consistent “container” data structures, e.g. stack of accounts, stack of points?

Dubious use of a container data structure:

a: ACCOUNT; p, q: POINT

point_stack: STACK ... ; account_stack: STACK ...

```
point_stack.put(p); account_stack.put(a)
q := point_stack.item
```

q.move(3.4, 1.2)

CONT 99-10

88

POSSIBLE APPROACHES TO THE CONSISTENT CONTAINER PROBLEM

- Ignore the problem until run-time. Possible “feature not available” errors. This is the dynamic typing approach of Smalltalk and, to a large extent, Java.
- Use a pseudo-universal type, such as “pointer to void” or “pointer to character” in C, to represent G in the class, and cast everything to that type.
- Duplicate code, manually or with the help of macro processor.
- Parameterize the class, giving a name G to the type of container elements; the features of the class may refer to type G. This is the Eiffel approach. (See also C++ templates.)

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

89

A GENERIC CLASS: RECONCILING FLEXIBILITY AND SAFETY

```
class STACK [G] feature
```

```
  put (x: G) is ...
```

```
  item: G is ...
```

```
end -- class STACK
```



FORMAL GENERIC PARAMETER

To use the class: obtain a GENERIC DERIVATION, e.g.

```
account_stack: STACK [ACCOUNT]
```



ACTUAL GENERIC PARAMETER

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

90

CONFORMANCE RULE

B [U] conforms to A [T] if and only if B is a descendant of A
and U conforms to T.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

91

USING GENERIC DERIVATIONS

```
account_stack: STACK [ACCOUNT]
```

```
point_stack: STACK [POINT]
```

```
a: ACCOUNT; p, q, r: POINT
```

```
...
```

```
point_stack.a put (p); point_stack.a put (q)
```

```
r := point_stack.a item; r.a move (3.0, -5.0)
```

```
account_stack.a put (a)
```

```
...
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

92

GENERICITY AND STATIC TYPING

Compiler will reject

```
point_stack.put (a)
```

```
account_stack.put (p)
```

To define more flexible data structures (e.g. stack of figures): use inheritance, polymorphism and dynamic binding.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

93

THE NOTION OF TYPING**IN AN OBJECT-ORIENTED CONTEXT**

An object-oriented language is statically typed if and only if it is possible to write a static checker which, if it accepts a system, guarantees that at run time, for any execution of a feature call $x.f$, the object attached to x (if any) will have at least one feature corresponding to f .

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

94

A GENERIC LIBRARY CLASS: ARRAYS

USING ARRAYS IN A CLIENT

```
a: ARRAY [REAL]
```

```
...
```

```
create a.make (1, 300)
```

```
a.put (3.5, 25)    -- (in Pascal: a[25] := 3.5)
```

```
x := a.item (i)    -- (in Pascal: x := a[i])
```

```
-- Alternatively:
```

```
x := a @ i        -- Using the function infix "@"
```

Also: **ARRAY2 [G]** etc.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

95

THE ARRAY CLASS

```
class ARRAY [G] creation
```

```
make
```

```
feature
```

```
lower, upper: INTEGER
```

```
count: INTEGER
```

```
make (min: INTEGER, max: INTEGER) is
```

```
-- Allocate array with bounds min and max
```

```
do ... end
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

96

THE ARRAY CLASS (CONTINUED)

```

item, infix "@" (i: INTEGER): G is
  -- Entry of index i
  require
    lower <= i; i <= upper
  do ... end

put (v: G; i: INTEGER) is
  -- Assign the value of v to the entry of index i.
  require
    lower <= i; i <= upper
  do ... end

invariant

  count = upper - lower + 1

end -- class ARRAY

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

97

GENERICITY + INHERITANCE 1: POLYMORPHIC DATA STRUCTURES

```

class STACK [G] feature
  ...
  item: G is ...
  put (x: G) is ...
end -- class STACK

fs: STACK [FIGURE]
t: TRIANGLE; p: POLYGON
c: CIRCLE; r: RECTANGLE

...

fs.put (p); fs.put (t); fs.put (s); fs.put (r); ...
fs.item.display

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

98

GENERICITY + INHERITANCE 2: CONSTRAINED GENERICITY

```

class
  VECTOR [G]
feature
  infix "+" (other: VECTOR [G]): VECTOR [G] is
    -- Sum of current vector and other
    require
      lower = other.lower; upper = other.upper
    local
      a, b, c: G
    do
      ... See next ...
    end
    ... Other features ...
end -- class VECTOR

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

99

CONSTRAINED GENERICITY (CONTINUED)

The body of infix "+":

```

create Result.make (lower, upper)
from i := lower until i > upper loop
  a := item (i)
  b := other.item (i)
  c := a + b      -- Requires a "+" operation on G!
  Result.put (c, i)
  i := i + 1
end

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

100

THE SOLUTION

Declare class **VECTOR** as

```
class VECTOR [G -> NUMERIC] feature
```

```
... The rest as before ...
```

```
end -- class VECTOR
```

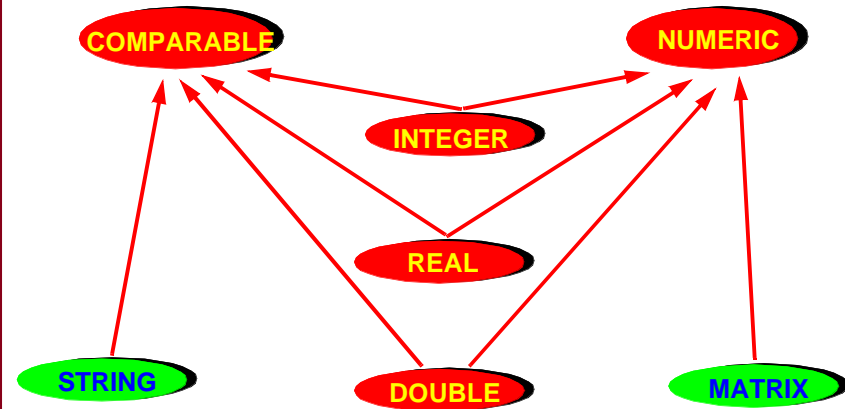
Class **NUMERIC** (from the Kernel Library) provides features infix "+", infix "*" and so on.

Better yet: make **VECTOR** itself a descendant of **NUMERIC**, effecting the corresponding features:

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

101

**MULTIPLE INHERITANCE:
COMBINING ABSTRACTIONS**

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

102

IMPROVING THE SOLUTION

Make **VECTOR** itself a descendant of **NUMERIC**, effecting the corresponding features:

```
class VECTOR [G -> NUMERIC] inherit
```

```
NUMERIC
```

```
feature
```

```
... The rest as before, including infix "+"...
```

```
end -- class VECTOR
```

Then it is possible to define e.g.

```
v: VECTOR [VECTOR [VECTOR [INTEGER]]]
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

103

CREATING GENERIC INSTANCES

```
class C [G -> CONSTRAINT create make end] feature
```

```
... x: C
```

```
... create x, make (...)
```

```
end
```

Procedure **make** must be a feature of **CONSTRAINT**, but not necessarily a creation procedure.

But the derivation **C [T]** is valid only if **T** defines **make** as one of its creation procedures.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

104

PROPERTIES OF GENERICITY

Applicable to all types, basic and user-defined:

- `LINKED_LIST [INTEGER]`
- `LINKED_LIST [EMPLOYEE]`

Can be nested: `LINKED_LIST [ARRAY [EMPLOYEE]]`

Can create generic instances

Constrained and unconstrained variants

Unconstrained form `C [G]` is equivalent to constrained form with `ANY`: `C [G → ANY]`

Conformance:

- `A [Y]` conforms to `A [X]` if `Y` conforms to `X`.
- `B [X]` conforms to `A [X]` if `B` conforms to `A`.
- (As a result) `B [Y]` conforms to `A [X]` if `Y` conforms to `X` and `B` to `A`.

CONT 99-10

105

ADDING GENERICITY TO JAVA

Numerous attempts: Pizza, Mitchell, Steele, Liskov, Thorup...

Major issues:

- Retrofitting: libraries, JVM...
- Static typing
- Handling basic types (int etc.) as well as programmer-defined types
- Constrained genericity
- Novariance
- Conformance
- Genericity + overloading
- Implementation efficiency
- Arrays

CONT 99-10

106

GJ GOALS (EXCERPTED)

- *Compatible with the Java language.* Every Java program and JVM class file is still legal and retains meaning.
- *Compatible with JVM.* Compiles into JVM code. No change to the JVM. Code can be executed on any JDK compliant browser.
- *Compatible with existing libraries.* One can compile a program that uses a parameterized type against existing class file libraries, such as the collections library.
- *Transparent translation.* Straightforward translation from GJ into Java. Translation leaves field and method names unchanged; easy to use reflection with GJ. Translation introduces minimal overhead.

CONT 99-10

107

GOALS (CONTINUED)

- *Efficient.* GJ is translated by erasure: no information about type parameters is maintained at run-time.
- *Futureproof.* Greater extendibility may be achieved by carrying type parameters at run-time, and this may be possible in future implementations.

CONT 99-10

108

COLLECTION CLASS IN GJ (FROM GJ TUTORIAL)

```

interface Collection<A> {
    public void add(A x);
    public Iterator<A> iterator();
}

interface Iterator<A> {
    public A next();
    public boolean hasNext();
}

class NoSuchElementException
extends RuntimeException {}

class LinkedList<A>
implements Collection<A> {
    protected class Node {
        public Iterator<A> iterator () {
            return new Iterator<A> () {
                protected Node ptr=head;
                public boolean hasNext () {
                    return ptr!=null; }
                public A next () {
                    if (ptr!=null) {
                        A elt=ptr.elt; ptr=ptr.next;
                        return elt;
                    } else
                        throw new
                            NoSuchElementException();
                }
            }
        }
    };
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

109

USING THE COLLECTION CLASS

```

class Test {
    public static void main
        (String[] args) {
        // Byte list
        LinkedList<Byte> xs =
            new LinkedList<Byte>();
        xs.add(new Byte(0));
        xs.add(new Byte(1));
        Byte x = xs.iterator().next();

        // String list
        LinkedList<String> ys =
            new LinkedList<String>();
        ys.add("zero");
        ys.add("one");
        String y = ys.iterator().next();

        // String list list
        LinkedList<LinkedList<String>>
            zss =
            new LinkedList<LinkedList<
                String>>();
        zss.add(ys);
        String z = zss.iterator().
            next().iterator().next();

        // String list treated as byte list
        Byte w = ys.iterator().next();
        // Compile-time error!
    }
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

110

A GENERIC METHOD

```

interface Comparator<A> {
    public int compare
        (A x, A y);
}

class ByteComparator
implements Comparator<Byte> {
    public int compare
        (Byte x, Byte y) {
        return x.byteValue()
            - y.byteValue();
    }
}

class Collections {
    public static <A> A max
        (Collection<A>
        xs, Comparator<A> c) {
        Iterator<A> xi =
            xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next(); ys.a
            if (c.compare(w,x) < 0)
                w = x;
        }
        return w;
    }
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

111

USING THE PARAMETRIC METHOD

```

class Test {
    public static YP void main
        (String[] args) {
        // Byte list with
        // byte comparator
        LinkedList<Byte> xs =
            new LinkedList<Byte>();
        xs.add(new Byte(0));
        xs.add(new Byte(1));
        Byte x = Collections.max
            (xs,
            new ByteComparator());

        // String list with
        // byte comparator
        LinkedList<String> ys =
            new LinkedList<String>();
        ys.add("zero");
        ys.add("one");
        String y =
            Collections.max(ys,
            new ByteComparator());
        // Run-time exception!
    }
}

```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

112

CONSTRAINED GENERICITY ("BOUNDS")

```
class Collections {
    public static <A implements Comparable<A>>
        A max (Collection<A> xs) {

    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0) w = x;
    }

    return w;
}
```

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

113

INTEGRATION WITH NON-GENERIC JAVA

Integrate generic classes with with non-generic ones through "erasure". Replace generic parameter by constraining type.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

114

TYPE INFERENCE

Use overloading to disambiguate various generic derivations.

Causes problems when the only disambiguation is from the result:

```
interface Iterator<A> {
    public boolean hasNext ();
    public A next ();
}
class Interval implements
Iterator<Integer> {
    private int i;
    private int n;

    public Interval (int k, int u)
        { i = k; n = u; }
    public boolean hasNext ()
        { return (i <= n); }
    public Integer next ()
        { return new Integer(i++); }
```

Generates two "next" functions, one returning an Object and one returning an Integer. Illegal in Java; OK at JVM level.

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

115

ISSUES

- Combination of overloading and genericity: ambiguities.
- Use of basic types.
- Complexity of type inference mechanism; clarity of error messages.
- Conformance rule: A [Y] does not conform to A [X]!
- Nonvariance.
- No creation of generic instances!

CONT 99-10

DESIGN BY CONTRACT AND GENERICITY FOR JAVA

116

THE LESSON

Fixing design flaws is nice.

Building things right is better.