



Design
by
Contract™

Design by Contract



A discipline of analysis, design, implementation,
management

Applications



Getting the software right

Analysis

Design

Implementation

Debugging

Testing

Management

Maintenance

Documentation

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.

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

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

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

With 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

With 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
```

The imperative and the applicative



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



With contracts

```
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



How not to do it

r(*i*: *INTEGER*): *BOOLEAN* is

require

$i \geq 0$ or $i < 0$

ensure

Result = true or *Result* = false

r(*x*: *BANK_ACCOUNT*): *BOOLEAN* is

require

$x \neq \text{Void}$ and $x.\text{balance} > 0$

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(...)*)

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.

Contracts and inheritance



Issues: what happens, under inheritance, to

- Class invariants?
- Routine preconditions and postconditions?

Invariants

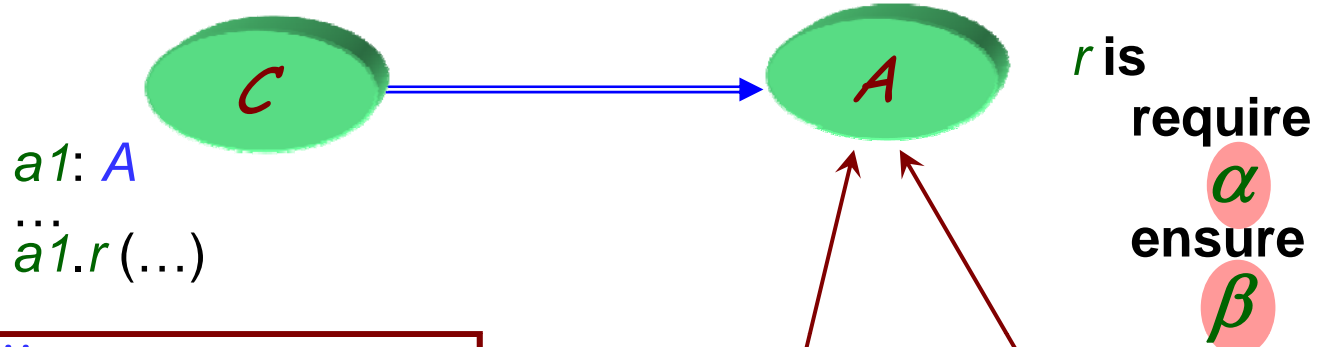


Invariant Inheritance rule:

- 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 in C:
  if  $a1.\alpha$  then
     $a1.r(\dots)$ 
    -- Here  $a1.\beta$  holds
  end
    
```

Client \Rightarrow

\Uparrow Inheritance

$^{++}$ Redefinition

Assertion redeclaration rule



When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition



Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

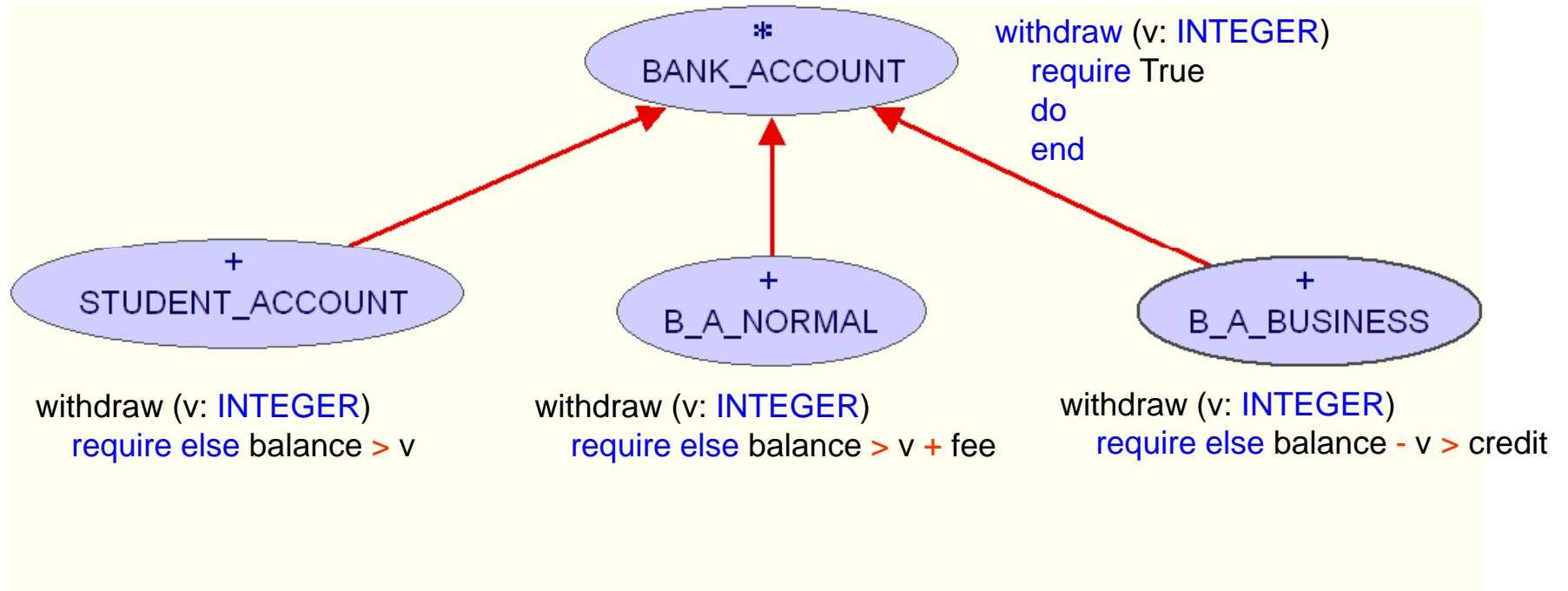
Redefined version may have nothing (assertions kept by default), or

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

Resulting assertions are:

- *original_precondition* **or** *new_pre*
- *original_postcondition* **and** *new_post*

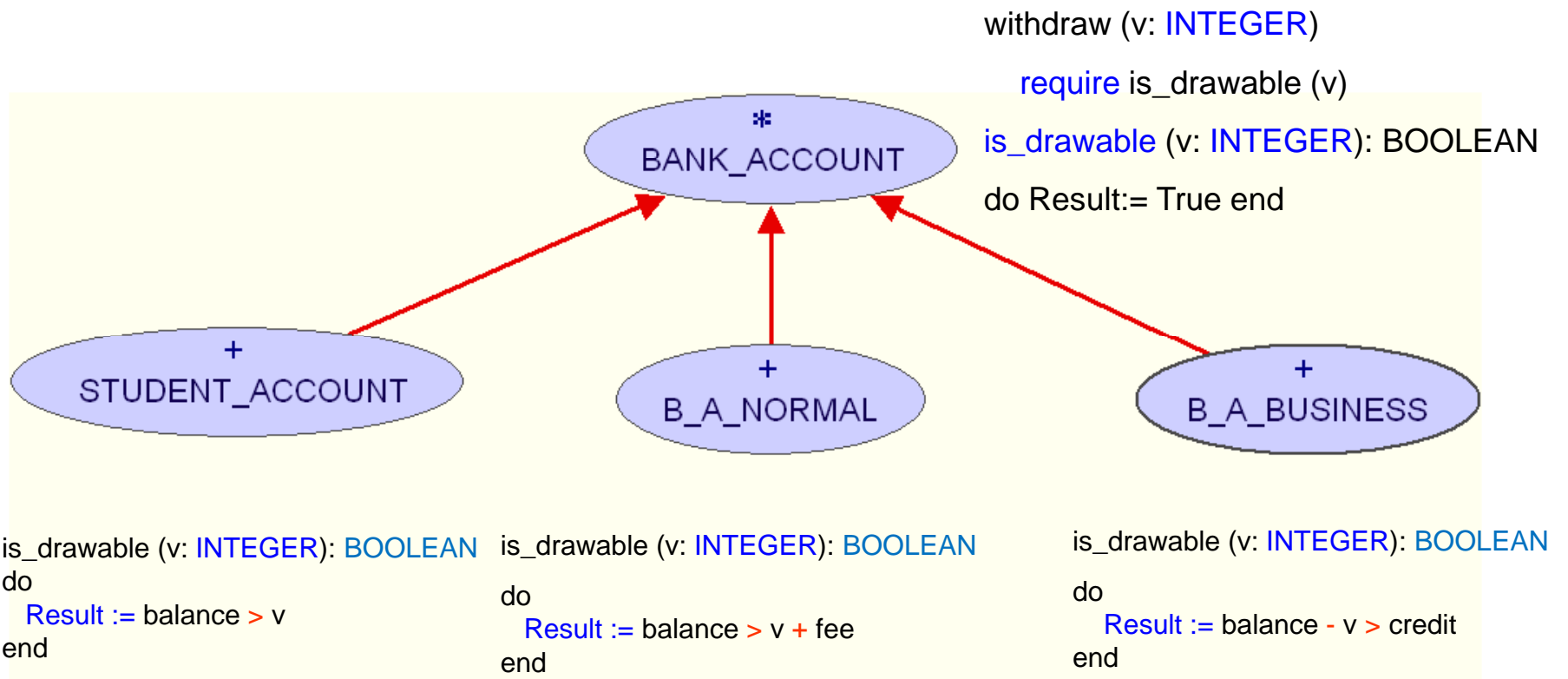
Example




Is the example correct?

↑ Inheritance

Example



 Inheritance

Contracts as a management tool



High-level view of modules for the manager:

- Follow what's going on without reading the code
- Enforce strict rules of cooperation between units of the system
- Control outsourcing