Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

Lecture 15: Design by Contract and exception handling
Topics

Part 1: Key concepts
Part 2: Contracts & documentation
Part 3: Contracts & testing
Part 4: Contracts & analysis, methodological notes
Part 5: Contracts & inheritance
Part 6: Contracts & loops
Part 7: Handling abnormal cases
Part 8: Contracts in various languages
Part 9: New developments
Part 10: Conclusion
Key concepts
Design by Contract

A discipline of analysis, design, implementation, management

Applications throughout the software lifecycle:
- Getting the software right: analysis, design, implementation
- Debugging & testing
- Automatic documentation
- Getting inheritance right
- Getting exception handling right
- Maintenance
- Management
Work on “axiomatic semantics“:

- R.W. Floyd (1967)
- E.W. Dijkstra (1978)

1970’s languages: CLU, Alphard
Eiffel (from 1985): connection with object technology

90s and onward: contract additions to numerous languages: C++, Java, C#, UML
Design by Contract

Every software element is intended to satisfy a certain goal, or *contract*

for the benefit of other software elements (and ultimately of human users)

The contract of any software element should be

- Explicit
- Part of the software element itself
The three questions

- What does it expect?
- What does it promise?
- What does it maintain?
Contracting components

Definition of what each element of the functionality:

- Expects (*precondition*)
- Promises (*postcondition*)
- Maintains (*invariant*)

Does not have to be complete (but wait)
What we do with contracts

- Write better software
- Analyze
- Design
- Reuse
- Implement
- Use inheritance properly
- Avoid bugs
- **Document** software automatically
- Help project managers do their job

(with run-time monitoring)

- Perform systematic testing
- Guide the debugging process
With and without contracts

```csharp
public virtual void Clear();
Removes all elements from the ArrayList.

Exceptions
<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotSupportedException</td>
<td>The ArrayList is read-only.</td>
</tr>
<tr>
<td></td>
<td>-or-</td>
</tr>
<tr>
<td></td>
<td>The ArrayList has a fixed size.</td>
</tr>
</tbody>
</table>

Remarks
Count is set to zero. Capacity remains unchanged.
```

`.Net collections library`

```
clear
-- Remove all items.

require
writable: not is_read_only
extendible: not is_fixed_size

do
-- Something

ensure
is_empty: count = 0
unchanged_capacity: capacity = old capacity

end
```

EiffelBase

with Karine Arnout (IEEE Computer)
Software construction consists of building 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.
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 \(Q\).
Hoare triples: a simple example

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

Most interesting properties:

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

“\(P\) is stronger than or equal to \(Q\)” means:

\(P\) implies \(Q\)

QUIZ: What is the strongest possible assertion? The weakest?
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
Software correctness (another quiz)

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. $\{\text{False}\}$ A $\{\ldots\}$
- 2. $\{\ldots\}$ A $\{\text{True}\}$
Properties of human 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
# A human contract

<table>
<thead>
<tr>
<th>deliver</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Bring package before 4 p.m.; pay fee.</td>
<td>(From postcondition:) Get package delivered by 10 a.m. next day.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Deliver package by 10 a.m. next day.</td>
<td>(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.</td>
</tr>
</tbody>
</table>
Properties of human 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
deferred class \textit{VAT} inherit \textit{TANK} \\

feature \\

\textit{in\_valve, out\_valve: VALVE} \\

\textit{fill} \\

\textit{-- Fill the vat.} \\

\textit{require} \\

\textit{in\_valve.open} \\

\textit{out\_valve.closed} \\

\textit{deferred} \\

\textit{ensure} \\

\textit{in\_valve.closed} \\

\textit{out\_valve.closed} \\

\textit{is\_full} \\

\textit{end} \\

\textit{empty, is\_full, is\_empty, gauge, maximum, ... [Other features] ...} \\

\textit{invariant} \\

\textit{is\_full = (gauge >= 0.97 \times maximum) and (gauge <= 1.03 \times maximum)} \\

\textit{end}
## Contracts for analysis

<table>
<thead>
<tr>
<th>fill</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure input valve is open, output valve closed</td>
<td>(From postcondition:) Get filled-up tank, with both valves closed</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Fill the tank and close both valves</td>
<td>(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position</td>
</tr>
</tbody>
</table>
“So, it’s like assert.h?”

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

Source: Reto Kramer
A class without contracts

class

ACCOUNT

feature -- Access

balance: INTEGER
  -- Balance

Minimum_balance: INTEGER = 1000
  -- Lowest permitted balance

feature {NONE} -- Deposit and withdrawal

add (sum: INTEGER)
  -- Add sum to the balance.
  do
    balance := balance + sum
  end
A class without contracts

feature -- Deposit and withdrawal operations

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

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

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

Value returned by function
class ACCOUNT
create make

feature {NONE} -- Initialization
make (initial_amount: INTEGER)
  -- Set up account with \textit{initial \_amount}.

\begin{verbatim}
require
  large\_enough: initial\_amount \geq Minimum\_balance

\end{verbatim}
do

\begin{verbatim}
  balance := initial\_amount
\end{verbatim}

\begin{verbatim}
ensure
  balance\_set: balance = initial\_amount
\end{verbatim}
end
Introducing contracts

feature -- Access

\[ balance: \text{INTEGER} \]
\[ \text{-- Balance} \]

\[ Minimum\_balance: \text{INTEGER} = 1000 \]
\[ \text{-- Lowest permitted balance} \]

feature \{NONE\} -- Implementation of deposit and withdrawal

\[ \text{add (sum: INTEGER)} \]
\[ \text{-- Add sum to the balance.} \]
\[ \text{do} \]
\[ \text{do} \]
\[ balance := balance + sum \]
\[ \text{ensure} \]
\[ \text{increased: } balance = \text{old balance} + sum \]
\[ \text{end} \]
**Introducing contracts**

**feature** -- Deposit and withdrawal operations

```plaintext
deposit (sum: INTEGER)
-- Deposit *sum* into the account.

require
not_too_small: sum >= 0

do
add (sum)

ensure
increased: balance = old balance + sum

end
```

**Precondition**

**Postcondition**
Introducing contracts

\textit{withdraw (sum: INTEGER)}

\begin{itemize}
  \item \textit{Withdraw sum from the account.}
\end{itemize}

\begin{verbatim}
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{verbatim}

\begin{itemize}
  \item \textbf{Precondition}
  \item \textbf{Postcondition}
\end{itemize}

\textbf{Value of balance, captured on entry to routine}
# The imperative and the applicative

<table>
<thead>
<tr>
<th>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\textit{balance} := \textit{balance} - \textit{sum})</td>
<td>(\textit{balance} = \textit{old balance} - \textit{sum})</td>
</tr>
</tbody>
</table>

## PRESCRIPTIVE

### What?
- Denotational
- Specification

### How?
- Operational
- Implementation
- Command
- Instruction
- Imperative

## DESCRIPTIVE

### What?
- Query
- Expression

### How?
- What?
### The contract

<table>
<thead>
<tr>
<th>withdraw</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure <em>sum</em> is neither too small nor too big</td>
<td>(From postcondition:) Get account updated with <em>sum</em> withdrawn</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Update account for withdrawal of <em>sum</em></td>
<td>(From precondition:) Simpler processing: may assume <em>sum</em> is within allowable bounds</td>
</tr>
</tbody>
</table>
Introducing contracts

\[ \text{may\_withdraw}(\text{sum}: \text{INTEGER}): \text{BOOLEAN} \]

\[ \text{-- Is it permitted to withdraw } \text{sum} \text{ from account?} \]

\[ \text{do} \]

\[ \text{Result} := (\text{balance} - \text{sum} \geq \text{Minimum\_balance}) \]

\[ \text{end} \]

\[ \text{invariant} \]

\[ \text{not\_under\_minimum}: \text{balance} \geq \text{Minimum\_balance} \]

\[ \text{end} \]
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: \( x.f(...) \)
The correctness of a class

For every creation procedure $cp$:

$\{\text{Pre}_{cp}\} \text{ do } _{cp} \{\text{INV and Post}_{cp}\}$

For every exported routine $r$:

$\{\text{INV and Pre}_{r}\} \text{ do } _{r} \{\text{INV and Post}_{r}\}$
Lists in EiffelBase

before

item

"Zurich"

index

after

count

 Cursor

back

forth

start

finish

1

count
Moving the cursor forward

- **before**
- **index**
- **Cursor**
- **forth**
- **"Zurich"**
- **after**
- **count**
Two queries, and command `forth`

```plaintext
feature -- Status report

    after: BOOLEAN
    -- Is there no valid cursor position to the right of cursor?

before: BOOLEAN
    -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

forth
    -- Move to next position; if no next position,
    -- ensure that `exhausted' will be true.

require -- from LINEAR
    not_after: not after
ensure then
    moved_forth: index = old index + 1
```
The contract language

Language of boolean expressions (plus old):

- No predicate calculus (i.e. no quantifiers, ∀ or ∃).
- Function calls permitted (e.g. in a STACK class):

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

remove
   -- Pop top of stack.
   require
   not is_empty
   do
      ...
   ensure
      not is_full
   end
```
The correctness of a class

For every creation procedure $cp$:

$\{\text{Pre}_{cp}\} \text{ do }_{cp} \{\text{INV and Post}_{cp}\}$

For every exported routine $r$:

$\{\text{INV and Pre}_{r}\} \text{ do }_{r} \{\text{INV and Post}_{r}\}$
A slightly more sophisticated version

\[
\text{balance} = \text{deposits}.\text{total} - \text{withdrawals}.\text{total}
\]
class ACCOUNT
create
make
feature \{NONE\} - Implementation

add (sum: INTEGER)
   -- Add sum to the balance.
   do
      balance := balance + sum
   ensure
      balance_increased: balance = old balance + sum
   end

deposits: DEPOSIT_LIST
withdrawals: WITHDRAWAL_LIST
feature {NONE} -- Initialization

make (initial_amount: INTEGER)
    -- 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 = 1000
    -- Minimum balance
New version

feature -- Deposit and withdrawal operations

deposit (sum: INTEGER)
  -- 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
New version

withdraw (sum: INTEGER)

-- 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
may_withdraw (sum: INTEGER): BOOLEAN
  -- 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
The correctness of a class

For every creation procedure \( cp \):

\[ \{ \text{Pre}_{cp} \} \text{ do}_{cp} \{ \text{INV and Post}_{cp} \} \]

For every exported routine \( r \):

\[ \{ \text{INV and Pre}_{r} \} \text{ do}_{r} \{ \text{INV and Post}_{r} \} \]
The new representation

\[
\text{balance} = \text{deposits.total} - \text{withdrawals.total}
\]
feature {NONE} - Initialization

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

What's wrong with this?
Design by contract: some applications

Getting the software right

Getting object-oriented development right: exceptions, inheritance...

Analysis and design

Automatic documentation

Project management

Maintenance

Testing and debugging
- 2 -

Contracts & documentation
Contracts for documentation

**Contract view** of a class: 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, invariant
- Remove any contract clause that refers to a secret feature

(This raises a problem; can you see it?)
class ACCOUNT
create make
feature {NONE} - Implementation

add(sum: INTEGER)
   -- Add sum to the balance.
   do
      balance := balance + sum
   ensure
      balance_increased: balance = old balance + sum
   end

deposits: DEPOSIT_LIST

withdrawals: WITHDRAWAL_LIST
The code (reminder)

feature {NONE} -- Initialization
   make (initial_amount: INTEGER)
       -- 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 = 1000
       -- Minimum balance
feature -- Deposit and withdrawal operations

\[
\text{deposit (sum: INTEGER)}
\]
\[
\text{-- Deposit sum into the account.}
\]

\[
\text{require}
\]
\[
\text{not too small: sum } \geq 0
\]

\[
\text{do}
\]
\[
\text{add (sum)
}\]

\[
\text{deposits.extend (create \{DEPOSIT\}.make (sum))}
\]

\[
\text{ensure}
\]
\[
\text{increased: balance = old balance + sum}
\]

\[
\text{one more: deposits.count = old deposits.count + 1}
\]

end
**The code (reminder)**

```plaintext
withdraw (sum: INTEGER)
   -- 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
```
The code (reminder)

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

invariant

  not_under_mininum: balance >= Minimum_balance

  consistent: balance = deposits.total - withdrawals.total

end
```
Contract view

class interface ACCOUNT create
    make
feature
    balance: INTEGER
        -- Balance

Minimum_balance: INTEGER = 1000
    -- Minimum balance

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

may_withdraw (sum: INTEGER): BOOLEAN
   -- Is it permitted to withdraw sum from the account?
   invariant
      not_under_minimum: balance >= Minimum_balance
end
Documenting a program

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

How to ensure that it doesn’t diverge from the code (the reverse Dorian Gray syndrome)?

The Single Product principle

The product is the software
Export rule for preconditions

In

```
feature
  r(...) require some_property
```

*some_property* must be exported!

No such requirement for postconditions and invariants.
Flat, interface

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

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

Interface view: the contract view of the flat view. Full interface documentation.
- 3 -

Contracts and testing
Contracts for testing

Contracts provide the right basis:
- A fault is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a fault:
- Precondition: in client
- Postcondition or invariant: in routine (supplier)

In EiffelStudio: select compilation option for contract monitoring at level of class, cluster or system.
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, such as if ... then ... else...

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

A DEFECT ("BUG")
Contracts: run-time effect

Compilation options (per class, in Eiffel):

- No assertion checking
- Preconditions only
- Preconditions and postconditions
- Preconditions, postconditions, class invariants
- All assertions
Contracts for testing and debugging

Contracts express implicit assumptions behind code

- A bug is a discrepancy between intent and code
- Contracts state the intent!

In EiffelStudio: select compilation option for run-time contract monitoring at level of:

- Class
- Cluster
- System

May disable monitoring when releasing software
A revolutionary form of quality assurance
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.
Lists in EiffelBase

before

"Zurich"

item

1

Cursor

back

forth

start

finish

index

after

count
Moving the cursor forward
Two queries, and command *forth*.

```plaintext
feature -- Status report

    after: BOOLEAN
        -- Is there no valid cursor position to the right of cursor?

    before: BOOLEAN
        -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

    forth
        -- Move to next position; if no next position,
        -- ensure that `exhausted' will be true.

    require -- from LINEAR
        not_after: not after

    ensure then
        moved_forth: index = old index + 1
```
Trying to insert too far right

(Already past last element!)
Where the cursor may go

Valid cursor positions

0 1

"Zurich"

index

count

count + 1

before

after
From the invariant of class LIST

```
invariant

prunable: prunable
before_definition: before = (index = 0)
after_definition: after = (index = count + 1)

-- from CHAIN
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

Valid cursor positions
Preconditions are particularly useful to find bugs in client code:

```java
your_list.insert(y, a + b + 1)
insert(x: G; i: INTEGER) {
    require
    i >= 0
    i <= count + 1
}
```
Next step: automated testing”

What can be automated:

- Test suite execution
- Resilience (continue test process after failure)
- Regression testing
- Test case generation
- Test result verification (*oracles*)
- Test extraction from failures
- Test case minimization

B. Meyer et al., *Programs that test themselves*, IEEE Computer, Sept. 2009
Contracts for testing

Contracts provide the right basis:

- A fault is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a fault:

- Precondition: in client
- Postcondition or invariant: in routine (supplier)

In EiffelStudio: select compilation option for contract monitoring at level of class, cluster or system.
Contracts & analysis, methodological notes
Precondition design

The client must *guarantee* the precondition before the call.
This does not necessarily mean *testing* for the precondition.

Scheme 1 (testing):

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

Scheme 2 (guaranteeing without testing):

```plaintext
my_stack.remove
...
my_stack.put(some_element)
```
Another example

\[ \sqrt{x, \text{epsilon: REAL}}: \text{REAL} \]

-- Square root of \( x \), precision \( \text{epsilon} \)

\begin{itemize}
  \item require \( x \geq 0 \)
  \item require \( \text{epsilon} > 0 \)
\end{itemize}

do ... 

ensure \( \text{abs} (\text{Result} \cdot 2 - x) \leq 2 \cdot \text{epsilon} \cdot \text{Result} \)

end
**The contract**

<table>
<thead>
<tr>
<th>sqrt</th>
<th><strong>OBLIGATIONS</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Provide non-negative value and precision that is not too small.</td>
<td>(From postcondition:) Get square root within requested precision.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Produce square root within requested precision.</td>
<td>(From precondition:) Simpler processing thanks to assumptions on value and precision.</td>
</tr>
</tbody>
</table>
Not defensive programming!

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

```plaintext
sqrt(x, epsilon : REAL): REAL
    -- 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
```
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.
class BYTECODE_PROGRAM feature
  verified: BOOLEAN

  trustful_execute (program: BYTECODE)
  require
    ok: verified
  do ... end

  distrustful_execute (program: BYTECODE)
  do
    verify
      if verified then trustful_execute (program)
  end
  end

  verify do ... end
end
How strong should a precondition be?

Two opposite styles:

- **Tolerant**: weak preconditions (including the weakest, \textit{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!
The demanding style

\[ \text{sqrt}(x, \text{epsilon}: \text{REAL}): \text{REAL} \]

\[ \text{-- Square root of } x, \text{ precision epsilon} \]

\[ \text{require} \]

\[ x \geq 0 \]

\[ \text{epsilon} > 0 \]

\[ \text{do} \]

\[ \ldots \]

\[ \text{ensure} \]

\[ \text{abs}(\text{Result}^2 - x) \leq 2 \times \text{epsilon} \times \text{Result} \]

\[ \text{end} \]
A tolerant style

\[ \text{sqrt}(x, \epsilon): \text{REAL} \]

\[
\begin{align*}
\text{require} & \quad \text{True} \\
\text{do} & \\
\text{if } x < 0 \text{ then} & \\
\quad & \quad \text{... Do something about it (?) ...} \\
\text{else} & \\
\quad & \quad \text{... Normal square root computation ...} \\
\quad & \quad \text{computed} := \text{True} \\
\text{end} \\
\text{ensure} & \quad \text{computed implies } \text{abs} \left( \text{Result}^2 - x \right) \leq 2 \times \epsilon \times \text{Result} \\
\text{end}
\end{align*}
\]
Contrasting styles

\[\text{put}(x: G)\]

-- Push \(x\) on top of stack.

\[\text{require} \quad \text{not } \text{is\_full}\]

\[\text{do} \quad \text{....} \quad \text{end}\]

\[\text{tolerant\_put}(x: G)\]

-- Push \(x\) if possible, otherwise set \text{impossible} to \text{True}.

\[\text{do} \quad \text{if not } \text{is\_full} \text{ then} \quad \text{put}(x)\]

\[\text{else} \quad \text{impossible} := \text{True}\]

\[\text{end} \quad \text{end}\]
Contracts and inheritance
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:

```plaintext
if a1.α then
    a1.r(…)
-- Here a1.β holds
end
```

++ Redefinition
Assertion redeclaration rule

When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition
Assertion redeclaration rule

A simple language rule does the trick!

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

\[
\text{require else } \text{new_pre} \\
\text{ensure then } \text{new_post}
\]

Resulting assertions are:

- \text{original_precondition or new_pre}
- \text{original_postcondition and new_post}
Contracts & loops
Quiz: what does this function compute?

\[ \text{euclid}(a, b \text{ INTEGER}) : \text{INTEGER} \]

-- Greatest common divisor of \( a \) and \( b \)

\text{require}
\begin{align*}
  a & > 0 ; b > 0 \\
\end{align*}

\text{local}
\begin{align*}
  m, n & : \text{INTEGER} \\
\end{align*}

\text{do}
\begin{align*}
  m & := a ; n := b \\
\end{align*}

\text{until}
\begin{align*}
  m & = n \\
\end{align*}

\text{loop}
\begin{align*}
  \text{if } m & > n \text{ then } \\
  m & := m - n \\
  \text{else } \\
  n & := n - m \\
\end{align*}

\text{end}

\text{end}

\text{Result} := m

\text{end}
Quiz: what does this function compute?

euclid (a, b: INTEGER): INTEGER
  -- Greatest common divisor of a and b
require
  a > 0 ; b > 0
local
  m, n: INTEGER
do
  from
    m := a ; n := b
invariant
  -- "????????"  
variant
  ???????
until
  m = n
loop
  if m > n then
    m := m - n
  else
    n := n - m
  end
end
Result := m
end
Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
    -- Greatest common divisor of a and b
require
    a > 0 ; b > 0
local
    m, n: INTEGER
do
    from
        m := a ; n := b
invariant
    -- gcd (m, n) = gcd (a, b)
variant
    ?????????
until
    m = n
loop
    if m > n then
        m := m - n
    else
        n := n - m
end
end
Result := m
end```

This function computes the greatest common divisor (GCD) of two integers, `a` and `b`. It uses the Euclidean algorithm, which repeatedly applies the operation of replacing the pair `(m, n)` with `(n, m - n)` or `(m - n, n)` (depending on which is larger) until the two numbers are equal, at which point the current value of `m` is the GCD of the original pair. The `require` clause specifies that both `a` and `b` must be positive integers, and the `invariant` clause ensures that the GCD of `m` and `n` remains constant throughout the algorithm.
Loop invariant

True after loop initialization
Preserved by loop body (i.e. if true before, will be true afterwards) when exit condition not true

from Init
until Exit
loop Body
end
Quiz: what does this function compute?

\[\text{euclid}(a, b: \text{INTEGER}): \text{INTEGER}\]

-- Greatest common divisor of \(a\) and \(b\)

require
\[a > 0 \; ; \; b > 0\]

local
\[m, n: \text{INTEGER}\]

do

from
\[m := a \; ; \; n := b\]

invariant

-- \(gcd(m, n) = gcd(a, b)\)

variant

?????????

until
\[m = n\]

loop

if \(m > n\) then
\[m := m - n\]
else
\[n := n - m\]
end

end

Result := \(m\)

end
Loop variant

Integer expression that must:

- Be non-negative when after initialization (from)

- Decrease (i.e. by at least one), while remaining non-negative, for every iteration of the body (loop) executed with exit condition not satisfied
Quiz: what does this function compute?

```plaintext
euclid(a, b: INTEGER): INTEGER
    -- Greatest common divisor of a and b
    require
        a > 0 ; b > 0
    local
        m, n: INTEGER
    do
        from
            m := a ; n := b
        invariant
            -- gcd(m, n) = gcd(a, b)
        variant
            max(m, n)
        until
            m = n
    loop
        if m > n then
            m := m - n
        else
            n := n - m
        end
    end
    Result := m
end
```
Invariants: loops as problem-solving strategy

A loop invariant is a property that:

- Is easy to **establish initially** (even to cover a trivial part of the data)
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**!
Computing the maximum of a list

from

invariant

across structure as i loop

Result := max(Result, i.item)

end
Reversing a list

from

pivot := first_element
first_element := Void

until pivot = Void loop

i := first_element
first_element := pivot
pivot := pivot.right
first_element.put_right(i)

end
Reversing a list

\[
\begin{align*}
\text{from} & \quad \text{pivot} := \text{first\_element} \\
& \quad \text{first\_element} := \text{Void} \\
\text{until} & \quad \text{pivot} = \text{Void} \quad \text{loop} \\
& \quad \i := \text{first\_element} \\
& \quad \text{first\_element} := \text{pivot} \\
& \quad \text{pivot} := \text{pivot}.\text{right} \\
& \quad \text{first\_element}.\text{put\_right}(\i) \\
\text{end}
\end{align*}
\]
Reversing a list

pivot := first_element
first_element := Void

until pivot = Void loop
i := first_element
first_element := pivot
pivot := pivot.right
first_element.put_right(i)
end

from
pivot := first_element
first_element := Void

right

1 2 3 4 5
Reversing a list

```
pivot := first_element
first_element := Void

until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
```
Reversing a list

```
pivot := first_element
first_element := Void

until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
```
Why does it work?

Invariant: from first_element following right, initial items in inverse order; from pivot, rest of items in original order

from pivot := first_element
first_element := Void
until pivot = Void loop
  i := first_element
  first_element := pivot
  pivot := pivot.right
  first_element.put_right(i)
end
- 6 -

Handling abnormal cases
An “abnormal case” is a case of applying a partial function outside of its domain

5 approaches:

- 1. A priori check
- 2. A posteriori check
- 3. Using agents
- 4. Return codes
- 5. Exception handling
Exception handling

Things not always happen in the ideal way!
Solution 1: Use standard control structures

```python
if not end_of_file then
    read_token
    if token /= "class" then
        message("File must start with class")
    else
        read_token
        if not token.is_identifier then
            message("Invalid class name")
        else
            if token.name.is_taken then
                message("Class name in use")
            else
                ...
```
Solution 1: a priori (check before)

\[
\begin{align*}
\text{if } y \cdot \text{property} \text{ then}\\
\quad a \cdot f(y)\\
\text{else}\\
\quad \ldots\\
\text{end}
\end{align*}
\]

\[
\begin{align*}
f(x: T) & \\
\quad \text{require} & \\
\quad x \cdot \text{property} & \\
\quad \text{do} & \\
\quad \ldots & \\
\quad \text{ensure} & \\
\quad \text{Result.} & \text{other_property}\\
\text{end}
\end{align*}
\]
Example: linear equation

Purpose: solve $A \times x = b$, given matrix $A$ and vector $b$ (the result $x$ will be a vector)

if $A$.regular then

    $x := A$.solution($b$)

else

    ...

end
Solution 1: a priori (check before)

\[
\text{if } y \cdot \text{property then}
\]

\[
a \cdot f(y)
\]

\[
\text{else}
\]

\[
\ldots
\]

\[
\text{end}
\]

\[
f(x: T) \quad \text{return}
\]

\[
\quad x \cdot \text{property}
\]

\[
\quad \text{do}
\]

\[
\quad \ldots
\]

\[
\quad \text{ensure}
\]

\[
\quad \text{Result, other_property}
\]

\[
\text{end}
\]
Solution 2: a posteriori (try and check)

\[ a \cdot \text{try}_f (y) \]

\[ \text{if } \text{it}_\text{worked} \text{ then} \]

\[ \text{... Continue normally ...} \]

\[ \text{else} \]

\[ \text{...} \]

\[ \text{end} \]

Solution 1:

\[ \text{if } y \cdot \text{property then} \]
\[ \quad a \cdot f (y) \]
\[ \text{else} \]
\[ \quad \text{...} \]
\[ \text{end} \]
Linear equation with solution 2

\[ A\cdot invert\,(b) \]

if \( A\cdot is\_inverted \) then

\[ x := A\cdot solution \]

... Continue normally ...

else

...

end

Solution 1:

if \( A\cdot regular \) then

\[ x := A\cdot solution\,(b) \]

else

...

end
Solution 3: using agents

Scheme 1:

\[ \text{action1} \]
\[ \text{if } ok1 \text{ then} \]
\[ \quad \text{action2} \]
\[ \quad \text{if } ok2 \text{ then} \]
\[ \quad \quad \text{action3} \]
\[ \quad \quad \text{-- More processing,} \]
\[ \quad \quad \text{-- more nesting ...} \]
\[ \text{end} \]
\[ \text{end} \]
Solution 4: return codes

```c
if (file_open(f)) {
    ... Continue with processing
}
else {
    ...
}
```
Solution 5: exceptions

In case of an abnormal situation:

- Interrupt execution
- Go up call chain
- If exception handler found, execute it
- Otherwise, program stops abnormally
What is an exception?

“An abnormal event”

Not a very precise definition

Informally: something that you don’t want to happen...
Exception vocabulary

- “Raise”, “trigger” or “throw” an exception
- “Handle” or “catch” an exception
try {
    ... Normal instructions, during which an exception may occur ...
} catch (ET1 e) {
    ... Handle exceptions of type ET1, details in e ...
} catch (ET2 e) {
    ... Handle exceptions of type ET2, details in e ...
}... Possibly more cases...
finally {
    ... Processing common to all cases, exception or not...
}
Java exceptions

Exceptions are objects, descendants of **Throwable**: 

![Exception Hierarchy Diagram]
Java: raising an exception

Instruction:

`throw my_exception`

The enclosing routine should be of the form

```java
my_routine (...) throws my_exception {
    ...
    if abnormal_condition
        throw my_exception;
}
```
How to use exceptions?

Two opposite styles:

- Exceptions as a control structure: Use an exception to handle all cases other than the most favorable ones (e.g. a key not found in a hash table triggers an exception)

- Exceptions as a technique of last resort
Exception handling

A formal basis:

- Introduce notion of contract
- The need for exceptions arises when a contract is broken by either of its parties (client, supplier)

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.
The original strategy

\[ r(\ldots) \]

require

\[ \ldots \]

do

\begin{align*}
\text{op_1} \\
\text{op_2} \\
\ldots \\
\text{op_i} \\
\ldots \\
\text{op_n}
\end{align*}

ensure

\[ \ldots \]

defer

\[ \ldots \]
Not going according to plan

\[ r(...) \]

**require**

... \[ \]

**do**

\[ op_1 \]

\[ op_2 \]

...

... \[ \]

**ensure**

... \[ \]

**end**

Fails, triggering an exception in \( r \) (\( r \) is *recipient* of exception).
Causes of exceptions in O-O programming

Three major kinds:

- Operating system signal: arithmetic overflow, no more memory, interrupt ...
- Assertion violation (if contracts are being monitored)
- Void call ($x.f$ with no object attached to $x$)

In Eiffel & Spec#, will go away
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 caller: “Organized Panic”
- Try again, using a different strategy (or repeating the same strategy): “Retrying”

(Rare third case: false alarm)
How not to do it

(From an Ada textbook)

```ada
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
```
The call chain

Routine call
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).
Transmitting over an unreliable line (1)

Max_attempts: INTEGER = 100

attempt_transmission (message: STRING)
    -- 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
Transmitting over an unreliable line (2)

Max_attempts: INTEGER = 100

failed: BOOLEAN

attempt_transmission(message: STRING)

-- 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
end
Another Ada textbook example

```ada
procedure attempt is begin
  <<Start>> -- Start is a label
  loop
    begin
      algorithm_1;
      exit; -- Alg. 1 success
    exception
      when others =>$>
        begin
          algorithm_2;
          exit; -- Alg. 2 success
        exception
          when others =>$>
            goto Start,
        end
    end
end main;
```

In Eiffel

```eiffel
atempt

local
  even: BOOLEAN

do
  if even then algorithm_2 else
    algorithm_1
  end

rescue
  even := not even; retry
end
```
Dealing with arithmetic overflow

\[\text{quasi\_inverse}(x: \text{REAL}): \text{REAL}\]

\[\quad \text{-- } 1/x \text{ if possible, otherwise } 0\]

local

\[\text{division\_tried}: \text{BOOLEAN}\]

do

if not \text{division\_tried} then

Result := 1/x

end

rescue

\[\text{division\_tried} := \text{True}\]

retry

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

\[ f(...) \begin{array}{l} do \\ \ldots \\ end \end{array} \]

is an abbreviation for

\[ f(...) \begin{array}{l} do \\ \ldots \\ rescue \\ -- \text{Nothing here} \\ end \end{array} \]

(This is a provisional rule; see next.)
The correctness of a class

For every creation procedure $cp$:

$$\{\text{Pre}_{cp}\} \text{ do } _{cp} \{\text{INV and Post}_{cp}\}$$

For every exported routine $r$:

$$\{\text{INV and Pre}_r\} \text{ do } _r \{\text{INV and Post}_r\}$$
Bank accounts

\[ balance := \text{deposits}\cdot\text{total} - \text{withdrawals}\cdot\text{total} \]

(A2)
Exception correctness

For the normal body:

\[
\begin{align*}
\{INV \text{ and } Pre_r, \} & \; \text{do}_r \{INV \text{ and } Post_r, \}
\end{align*}
\]

For the exception clause:

\[
\begin{align*}
\{???\} & \; \text{rescue}_r \{???\}
\end{align*}
\]
Exception correctness

For the normal body:

\[
\{ \text{INV and Pre}_r \} \text{ do } \{ \text{INV and Post}_r \}
\]

For the exception clause:

\[
\{ \text{True} \} \text{ rescue } \{ \text{INV} \}
\]
Absence of a rescue clause is equivalent to a default rescue clause:

\[
\begin{align*}
\text{f}(\ldots) & \quad \text{do} \\
& \quad \ldots \\
\text{end}
\end{align*}
\]

is an abbreviation for

\[
\begin{align*}
\text{f}(\ldots) & \quad \text{do} \\
& \quad \ldots \\
& \quad \text{rescue} \quad \text{default_rescue} \\
\text{end}
\end{align*}
\]

The task of \textit{default_rescue} is to restore the invariant.
For finer-grain exception handling

Every exception has a type, a descendant of the library class `EXCEPTION`

Query `last_exception` gives an object representing the last exception that occurred

Some features of class `EXCEPTION`:

- `name`
- `is_assertion_violation`, etc.
- `raise`
Another challenge today

Exceptions in a concurrent world

What if the call chain is no longer available?
Exception handling: summary and conclusion

Exceptions as a control structure *(internally triggered)*:

- Benefits are dubious at best
- An exception mechanism is needed for unexpected *external* events
- Need precise methodology; must define what is “normal” and “abnormal”. Contracts provide that basis.
- Next challenge is concurrency & distribution
Design by Contract in various languages
What we do with contracts

Write better software
Analyze
Design
Reuse
Implement
Use inheritance properly
Avoid bugs
Document software automatically
Help project managers do their job

(with run-time monitoring)
Perform systematic testing
Guide the debugging process
Emulating Design by Contract mechanisms

Basic step (programmer discipline):

- Add preconditions and postconditions
- Use switch to turn monitoring on or off
- Help for analysis, methodology, debugging, but
  - No documentation help
  - No class invariants
  - No connection with O-O structure
  - No inherited assertions
  - No connection with exception handling

Other techniques:

- Macros (C, C++)
- Language extensions, e.g. preprocessor
The macro approach

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

Set of C++ macros and commands for gdb debugger. Replaces assert.h.

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”).
void intsqrt(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);
    DI(((r * r) <= $r) && ($r < (r + 1) * (r + 1)));
    /* use $r in postcondition */
}
OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.


“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.”
Java Modeling Language (JML)

Contract-equipped extension for Java

Assertions are in the form of JavaDoc comments

Rich tool suite for tests and proofs
public class BankingExample {
    public static final int MAX_BALANCE = 1000;
    private /*@ spec_public @*/ int balance;
    private /*@ spec_public @*/ boolean isLocked = false;
    //@ public invariant balance >= 0 && balance <= MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() { balance = 0; }

    //@ requires 0 < amount && amount + balance < MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == old(balance + amount);
    public void credit(int amount) { balance += amount; }
}
//@ requires 0 < amount && amount <= balance;
//@ assignable balance;
//@ ensures balance == \old(balance) - amount;
public void debit(int amount) { balance -= amount; }

//@ ensures isLocked == true;
public void lockAccount() { isLocked = true; }

//@ requires !isLocked;
//@ ensures \result == balance;
//@ also
//@ requires isLocked;
//@ signals_only BankingException;
public /*@ pure @*/ int getBalance() throws BankingException {
    if (!isLocked) { return balance; }
    else { throw new BankingException(); }
}
Object Constraint Language

Contract extension to UML
Includes support for:

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

Postconditions:

\[ \text{post: result} = \text{collection} \rightarrow \text{iterate} \]
\[ (\text{elem}; \text{acc} : \text{Integer} = 0 | \text{acc} + 1) \]

\[ \text{post: result} = \text{collection} \rightarrow \text{iterate} \]
\[ (\text{elem}; \text{acc} : \text{Integer} = 0 | \]
\[ \text{if elem} = \text{object} \text{ then acc} + 1 \text{ else acc endif}) \]

\[ \text{post: T.allInstances} \rightarrow \text{forAll} \]
\[ (\text{elem} | \text{result} \rightarrow \text{includes}(\text{elem}) = \text{set} \rightarrow \]
\[ \text{includes}(\text{elem}) \text{ and set2} \rightarrow \text{includes}(\text{elem})) \]
Spec#

Contract-equipped version of C# language

Originally developed at Microsoft Research

Includes non-null types
static int min (int x, int y)
    requires 0 <= x && 0 <= y ;
    ensures  x < y ? result == x: result == y;
{
    int m;
    if (x < y)
        m = x;
    else
        m = y;
    return m;
}
The Spec# verifier

Spec#

Spec# compiler

MSIL ("bytecode")

Translator

Boogie language

Inference engine

V.C. generator

SMT solver (Z3)

verification conditions

"correct" or list of errors

Source: Rustan Leino
Code contracts

Introduced in 2009 to provide a "language-agnostic way to express coding assumptions in .NET programs" (Microsoft)

Set of static library methods for writing preconditions, postconditions, and "object invariants", with tools:

- ccrewrite to generate run-time checking
- cccheck: static checker
- ccdoc: for documentation

Applied to large part of mscore library
New developments
The next steps

Pushing some properties to the type system:
- Void safety

More expressive specifications

Concurrency

Proofs
Concurrent mechanism:

- General object-oriented notation for concurrent programs

- Based on reinterpretation of contracts: preconditions become wait conditions
put \((b: \text{BUFFER}[G]; v: G)\)

--- Store \(v\) into \(b\).

require

not \(b.\text{is_full}\)

do ...

ensure

not \(b.\text{is_empty}\)
end

my_queue : \text{BUFFER}[T]

...

if not my_queue.is_full then

put (my_queue, t)

end
Increasing expressive power

Components to prove (e.g. EiffelBase)

Eiffel Model Library
Classes correspond to mathematical concepts:

\[
\text{SET}[G], \text{FUNCTION}[G, H], \text{TOTAL\_FUNCTION}[G, H], \text{RELATION}[G, H], \text{SEQUENCE}[G], \ldots
\]

Completely applicative: no attributes (fields), no implemented routines (all completely deferred)

Specified with contracts (unproven) reflecting mathematical properties

Expressed entirely in Eiffel
Example MML class

class \textit{SEQUENCE}[G] \text{ feature} \\
\hspace{1em} \textit{count} : \textit{NATURAL} \hspace{1em} \text{-- Number of items} \\
\hspace{1em} \textit{last} : G \hspace{1em} \text{-- Last item} \\
\hspace{1em} \textit{extended}(x) : \textit{SEQUENCE}[G] \hspace{1em} \text{-- Identical sequence except } x \text{ added at end.} \\
\hspace{2em} \text{ensure} \\
\hspace{3em} \textit{Result.} \textit{count} = \textit{count} + 1 \\
\hspace{3em} \textit{Result.} \textit{last} = x \\
\hspace{3em} \textit{Result.} \textit{sub}(1, \textit{count}) \sim \textit{Current} \\
\\n\hspace{1em} \textit{mirrored} : \textit{SEQUENCE}[G] \hspace{1em} \text{-- Same items in reverse order.} \\
\hspace{2em} \text{ensure} \\
\hspace{3em} \textit{Result.} \textit{count} = \textit{count} \\
\hspace{3em} \text{...}
class LINKED_LIST [G]

feature ...

remove_front

-- Remove first item.
require not empty
do
  first := first.right
ensure
  model = old model.tail
count = old count - 1
end
  first = old item (2)
end ...

...
Principles

Very simple mathematics only

- Logic
- Set theory
EiffelBase2

In progress: library of fully specified (MML) classes, covering fundamental data structures and algorithms, and designed for verification: tests and proofs
Verification As a Matter Of Course

AutoProof

Inter. prover

Alias analysis

Sep. logic prover

Invariant inference

EVE

Suggestions

Arbiter

AutoTest

Test execution

Test case generation

Invariant inference

AutoFix

Test results
Conclusion
Design by Contract: technical benefits

More focused process: writing to spec

Sound basis for reuse

Exception handling guided by precise definition of "normal" and "abnormal" cases

Interface documentation automatically generated, up-to-date, can be trusted

Faults occur close to cause, found faster & more easily

Guide for black-box test case generation.
Design by Contract: 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
“I believe that the use of Eiffel-like module contracts is the most important non-practice in software today”