Software architecture

Bertrand Meyer

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

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

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

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

The Single Product principle

The product is the software
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.
Ariane 5, 1996

$500 million, not insured.

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

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

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

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

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

**deliver**

<table>
<thead>
<tr>
<th></th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:)</td>
<td>(From postcondition:)</td>
</tr>
<tr>
<td></td>
<td>Bring package before 4 p.m.; pay fee.</td>
<td>Get package delivered by 10 a.m. next day.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:)</td>
<td>(From precondition:)</td>
</tr>
<tr>
<td></td>
<td>Deliver package by 10 a.m. next day.</td>
<td>Not required to do anything if package delivered after 4 p.m., or fee not paid.</td>
</tr>
</tbody>
</table>
A view of software construction

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

- These definitions are the contracts.
Properties of contracts

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

**deferred class**

*PLANE*

**feature**

*start_take_off is*  
-- Initiate take-off procedures.

**require**

controls.passed  
assigned_runway.is_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 <= 20) implies (assigned_runway.owner = Current)  
moving = (speed > 10)

**end**

---

*Chair of Software Engineering*  
Software architecture
deferred class

    TANK

feature

    in_valve, out_valve: VALVE
    fill is
    require
    in_valve.open
    out_valve.is_closed
    deferred
    ensure
    in_valve.is_closed
    out_valve.is_closed
    is_full
    end

    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...

invariant

    is_full = (gauge >= 0.97 * maximum) and (gauge <= maximum)

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><em>(Satisfy precondition:)</em> Make sure input valve is open, output valve is closed.</td>
<td><em>(From postcondition:)</em> Get filled-up tank, with both valves closed.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td><em>(Satisfy postcondition:)</em> Fill the tank and close both valves.</td>
<td><em>(From precondition:)</em> Simpler processing thanks to assumption that valves are in the proper initial position.</td>
</tr>
</tbody>
</table>
Correctness in software

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

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

\[
\{P\} \ A \ \{Q\}
\]

- “Hoare triple”

- What this means (total correctness):
  - Any execution of \(A\) started in a state satisfying \(P\) will terminate in a state satisfying \(Q\).
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?
Specifying a square root routine

{x \geq 0}

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

{\text{abs } (y^2 - x) \leq 2 \times \text{epsilon} \times y}

-- i.e.: y approximates exact square root of x

-- within epsilon
Software correctness

- Consider
  \[\{P\} A \{Q\}\]

- Take this as a job ad in the classifieds.

- Should a lazy employment candidate hope for a weak or strong \(P\)? What about \(Q\)?

- Two special offers:
  1. \(\{False\} A \{\ldots\}\)
  2. \(\{\ldots\} A \{True\}\)
A contract (from EiffelBase)

\[\text{extend } (\text{new: } G; \text{key: } H)\]

\(--\text{ Assuming there is no item of key key,}\)
\(--\text{ insert new with key; set inserted.}\)

\text{require}

\text{key\_not\_present: not has (key)}

\text{ensure}

\text{insertion\_done: item (key) = new}
\text{key\_present: has (key)}
\text{inserted: inserted}
\text{one\_more: count = old count + 1}
## The contract

<table>
<thead>
<tr>
<th>Routine</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td><strong>PRECONDITION</strong></td>
<td><strong>POSTCONDITION</strong></td>
</tr>
<tr>
<td>Supplier</td>
<td><strong>POSTCONDITION</strong></td>
<td><strong>PRECONDITION</strong></td>
</tr>
</tbody>
</table>
A class without contracts

class ACCOUNT

feature -- Access

  balance: INTEGER
     -- Balance

  Minimum_balance: INTEGER is 1000
     -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

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

**feature** -- Deposit and withdrawal operations

```plaintext

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
```

Chair of Software Engineering

Software architecture
Introducing contracts

```plaintext
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
feature -- Deposit and withdrawal operations

\[\text{deposit (sum: INTEGER) is}\]
\[\text{-- Deposit sum into the account.}\]
\[\text{require} \]
\[\text{not too small: sum} \geq 0\]
\[\text{do}\]
\[\text{add (sum)}\]
\[\text{ensure}\]
\[\text{increased: balance = old balance + sum}\]
\[\text{end}\]
Introducing contracts

withdraw \((\text{sum}: \text{INTEGER})\) is
-- Withdraw \text{sum} from the account.

\textbf{require}
\begin{align*}
\text{not\_too\_small: } \text{sum} &\geq 0 \\
\text{not\_too\_big: } &\sum \leq \text{balance} - \text{Minimum\_balance}
\end{align*}

\textbf{do}
\begin{align*}
\text{add} \ (\ - \text{sum}) \\
\text{-- i.e. } \text{balance} := \text{balance} - \text{sum}
\end{align*}

\textbf{ensure}
\begin{align*}
\text{decreased: } \text{balance} = \text{old balance} - \text{sum}
\end{align*}

\textbf{end}
## 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>
**The imperative and the applicative**

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

<table>
<thead>
<tr>
<th>PRESCRIPTIVE</th>
<th>DESCRIPTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>How?</td>
<td>What?</td>
</tr>
<tr>
<td>Operational</td>
<td>Denotational</td>
</tr>
<tr>
<td>Implementation</td>
<td>Specification</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
</tr>
<tr>
<td>Instruction</td>
<td>Expression</td>
</tr>
<tr>
<td>Imperative</td>
<td>Applicative</td>
</tr>
</tbody>
</table>
Introducing contracts

\[\text{may\_withdraw} \ (\text{sum}: \text{INTEGER}) : \text{BOOLEAN is} \]
\[\quad \text{-- Is it permitted to withdraw sum from the} \]
\[\quad \text{-- account?} \]
\[\text{do} \]
\[\quad \text{Result} := (\text{balance} - \text{sum} \geq \text{Minimum\_balance}) \]
\[\text{end} \]

\[\text{invariant} \]
\[\quad \text{not\_under\_minimum}: \text{balance} \geq \text{Minimum\_balance} \]
\[\text{end} \]
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 \( cp \):
  \[
  \{ \text{Pre}_{cp} \} \text{do}_{cp} \{ \text{Post}_{cp} \text{ and } \text{INV} \}
  \]

- For every exported routine \( r \):
  \[
  \{ \text{INV and Pre}_r \} \text{do}_r \{ \text{Post}_r \text{ and } \text{INV} \}
  \]
Uniform Access

\[ \text{balance} = \text{deposits}.\text{total} - \text{withdrawals}.\text{total} \]
A (slightly) more sophisticated version

class

    ACCOUNT

create

    make

feature {NONE} -- Implementation

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

    deposits: DEPOSIT_LIST

    withdrawals: WITHDRAWAL_LIST
New version

feature \{NONE\} -- Initialization

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

feature -- Access

balance: INTEGER
   -- Balance
Minimum_balance: INTEGER is 1000
   -- Minimum balance
**feature** -- Deposit and withdrawal operations

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

*Chair of Software Engineering*
withdraw \( (sum: INTEGER) \) is

-- Withdraw \( sum \) from the account.

require

not\_too\_small: \( sum \geq 0 \)
not\_too\_big: \( sum \leq balance - Minimum\_balance \)

do

\textit{add} \(- sum\)

withdrawals.extend (\textit{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
   -- Is it permitted to withdraw sum from the
   -- account?
   do
      Result := (balance - sum >= Minimum_balance)
   end

invariant
   not_under_minimum: balance >= Minimum_balance
   consistent: balance = deposits.total - withdrawals.total
end
The correctness of a class

- For every creation procedure \( cp \):
  \[
  \{ \text{Pre}_{cp} \} \ do_{cp} \ {\text{Post}_{cp} \ and \ INV}\]

- For every exported routine \( r \):
  \[
  \{ INV \ and \ Pre_r \} \ do_r \ {Post_r \ and \ INV}\]
feature \{NONE\} -- Initialization

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

feature {NONE} -- Initialization

make (initial_amount: INTEGER) is
  -- Set up account with initial_amount.
  require
  large_enough: initial_amount >= Minimum_balance
  do

  create deposits.make
  create withdrawals.make

  deposit (initial_amount)

  ensure
  balance_set: balance = initial_amount

end
Contracts: run-time effect

- Compilation options (per class, in Eiffel):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions
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. \texttt{if ... then ... else...}).

- A run-time assertion violation is something else: the manifestation of a \textsc{DEFECT} (“\textsc{BUG}”).
The contract language

- **Language of boolean expressions** (plus `old`):
  - No predicate calculus (i.e. no quantifiers, $\forall$ or $\exists$).
  - Function calls permitted (e.g. in a `STACK` class):

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

remove (x: G) is  
    -- Pop top of stack.
    require
        not is_empty
    do
        ...
    ensure
        not is_full
end
```
The contract language

- First order predicate calculus may be desirable, but not sufficient anyway.
- Example: “The graph has no cycles”.
- In assertions, use only side-effect-free functions.
- Use of iterators provides the equivalent of first-order predicate calculus in connection with a library such as EiffelBase or STL. For example (Eiffel agents, i.e. routine objects):

```eiffel
do
    Result := (x > 0)
end
```

with

```eiffel
is_positive (x: INTEGER): BOOLEAN is
    do
        Result := (x > 0)
    end
```
### The imperative and the applicative

<table>
<thead>
<tr>
<th>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance := balance - sum</td>
<td>balance = old balance - sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prescriptive</th>
<th>Descriptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>How?</td>
<td>What?</td>
</tr>
<tr>
<td>Operational</td>
<td>Denotational</td>
</tr>
<tr>
<td>Implementation</td>
<td>Specification</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
</tr>
<tr>
<td>Instruction</td>
<td>Expression</td>
</tr>
<tr>
<td>Imperative</td>
<td>Applicative</td>
</tr>
</tbody>
</table>
What are contracts good for?

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

- Quality assurance, testing, debugging (especially in connection with the use of libraries).
- Exception handling.
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. \texttt{if ... then ... else...}).

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

\begin{center}
A DEFECT (“BUG”)
\end{center}
Contracts and quality assurance

- Precondition violation: Bug in the client.
- Postcondition violation: Bug in the supplier.
- Invariant violation: Bug in the supplier.

\{P\} A \{Q\}
Contracts and bug types

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

```latex
your_list.insert (y, a + b + 1)

class LIST [G]
...
insert (x: G; i: INTEGER) is
  require
  i >= 0
  i <= count + 1
```
Contracts and quality assurance

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

- Compilation options (reminder):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions
Contracts missed

- Ariane 5 (see Jézéquel & Meyer, IEEE Computer, January 1997)
- Lunar Orbiter Vehicle
- Failure of air US traffic control system, November 2000
- Y2K
- etc. etc. etc.
Contracts and quality assurance

- Contracts enable QA activities to be based on a precise description of what they expect.
- Profoundly transform the activities of testing, debugging and maintenance.

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

Tom de Marco, IEEE Computer, 1997
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.
Recall example class:

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

make \((\text{initial\_amount}: \text{INTEGER})\) is
  -- Set up account with \text{initial\_amount}.
  require large\_enough: \text{initial\_amount} \geq \text{Minimum\_balance}
  do
    deposit \((\text{initial\_amount})\)
    create deposits.make
    create withdrawals.make
  ensure balance\_set: balance = \text{initial\_amount}
end

feature -- Access

balance: \text{INTEGER}
  -- Balance

Minimum\_balance: \text{INTEGER} is 1000
  -- Minimum balance
Class example (continued)

feature -- Deposit and withdrawal operations

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

withdraw \( (\text{sum}: \text{INTEGER}) \) is
   -- Withdraw sum from the account.
   \textbf{require}
   not\_too\_small: \text{sum} \geq 0
   not\_too\_big: \text{sum} \leq \text{balance} - \text{Minimum\_balance}
   \textbf{do}
   \text{add} (\text{- sum})
   \text{withdrawals.extend} (\text{create} \{\text{WITHDRAWAL}\}.\text{make} (\text{sum}))
   \textbf{ensure}
   decreased: \text{balance} = \text{old balance} - \text{sum}
   one\_more: \text{withdrawals.count} = \text{old withdrawals.count} + 1
end
may_withdraw (sum: INTEGER): BOOLEAN is
    -- Is it permitted to withdraw sum from the
    -- account?
    do
        Result := (balance - sum >= Minimum_balance)
    end

invariant

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

end
Contract form: Definition

- Simplified form of class text, retaining interface elements only:
  - Remove any non-exported (private) feature.

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

- In

```plaintext
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.
class interface ACCOUNT

create

  make

feature

  balance: INTEGER
    -- Balance

  Minimum_balance: INTEGER is 1000
    -- Minimum balance

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

```plaintext
withdraw (sum: INTEGER)
  -- Withdraw sum from the account.
  require
    notTooSmall: sum >= 0
    notTooBig: sum <= balance - Minimum_balance
  ensure
    decreased: balance = old balance - sum
    oneMore: withdrawals.count = old withdrawals.count + 1

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

invariant

  notUnderMinimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total

end
```

*Chair of Software Engineering*
Flat, interface

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

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

- **Interface form**: the contract form of the flat form. Full interface documentation.
Uses of the contract and interface forms

- Documentation, manuals
- Design
- Communication between developers
- Communication between developers and managers
The contract form — i.e. the set of contracts governing a class — should be the standard form of library documentation.

Reuse without a contract is sheer folly.

See the Ariane 5 example.
Contracts and inheritance

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

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

Assertion redeclaration rule

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

- Redeclaration covers both redefinition and effecting.

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

  \[ n \ > \ 1 \]

  implies (is stronger) than

  \[ n \ ^{26} + 3 \ * \ n \ ^{25} \ > \ 3 \]
Assertion redeclaration rule in Eiffel

- A simple language rule does the trick!
- Redefined version may **not** have `require` or `ensure`.
- May have nothing (assertions kept by default), or

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

- Resulting assertions are:
  - `new_pre` or else `original_precondition`
  - `original_postcondition` and then `new_post`
deferred class \textit{LIST} [G]

inherit

\textit{CHAIN} [G]

feature

\textit{has} (x: G): BOOLEAN is

-- Does x appear in list?

do

from

start

until

after or else found (x)

loop

forth

end

Result := not after

end
Sequential structures

before

item

after

1

start

back

forth

index

count
Sequential structures (continued)

forth is
require
not after
defered
ensure
\[ index = \text{old} \ index + 1 \]
end

start is
defered
ensure
\[ \text{empty or else} \ index = 1 \]
end
Sequential structures (continued)

\[ \text{index: INTEGER is} \]
\[
\text{deferred end} \\
\]

\[ \ldots \text{empty, found, after, ...} \]

\textbf{invariant}

\[ 0 \leq \text{index} \]
\[ \text{index} \leq \text{size} + 1 \]

\[ \text{empty implies (after or before)} \]

\textbf{end}
Descendant implementations

- **CHAIN**
  - has+

- **LIST**
  - has+
  - after*
  - forth*
  - item*
  - start*

- **ARRAYED_LIST**
  - after+
  - forth+
  - item+
  - start+

- **LINKED_LIST**
  - after+
  - forth+
  - item+
  - start+

- **BLOCK_LIST**
  - after+
  - forth+
  - item+
  - start+
## Implementation variants

<table>
<thead>
<tr>
<th></th>
<th>start</th>
<th>forth</th>
<th>after</th>
<th>found (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrayed list</strong></td>
<td>i := 1</td>
<td>i := i + 1</td>
<td>i &gt; count</td>
<td>t[i] = x</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>c := first_cell</td>
<td>c := c.right</td>
<td>c := Void</td>
<td>c.item = x</td>
</tr>
<tr>
<td><strong>File</strong></td>
<td>rewind</td>
<td>read</td>
<td>end_of_file</td>
<td>f↑ = x</td>
</tr>
</tbody>
</table>
Methodological notes

- Contracts are not input checking tests...
- ... but they can be used to help weed out undesirable input.

Filter modules:

- External objects
- Input and validation modules
- Processing modules

Preconditions here only
Precondition design

- The client must **guarantee** the precondition before the call.
- This does not necessarily mean **testing** for the precondition.
- Scheme 1 (testing):

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

- Scheme 2 (guaranteeing without testing):

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

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

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

\textbf{require}

\[
x \geq 0 \\
\text{epsilon} \geq 0
\]

\textbf{do}

\[
\ldots
\]

\textbf{ensure}

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

\textbf{end}
# The contract

<table>
<thead>
<tr>
<th>sqrt</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td><em>(Satisfy precondition:)</em> Provide non-negative value and precision that is not too small.</td>
<td><em>(From postcondition:)</em> Get square root within requested precision.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td><em>(Satisfy postcondition:)</em> Produce square root within requested precision.</td>
<td><em>(From precondition:)</em> Simpler processing thanks to assumptions on value and precision.</td>
</tr>
</tbody>
</table>
It is never acceptable to have a routine of the form

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

Not defensive programming!
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) is
    require
      ok: verified
    do
      ...
    end
  distrustful_execute (program: BYTECODE) is
    do
      verify
      if verified then
        trustful_execute (program)
      end
    end
  verify is
    do
      ...
    end
end
How strong should a precondition be?

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

- Partly a matter of taste.

- But: demanding style leads to a better distribution of roles, provided the precondition is:
  - Justifiable in terms of the specification only.
  - Documented (through the short form).
  - Reasonable!
sqrt \((x, \epsilon)\): \(\text{REAL}\) is
\[
-- \text{Square root of } x, \text{ precision } \epsilon
-- \text{Same version as before}
\]
\begin{verbatim}
require
\[
x \geq 0 \quad \epsilon \geq 0
\end{verbatim}
do
\[
\ldots
\]
ensure
\[
\text{abs} (\text{Result} ^ 2 - x) \leq 2 \times \epsilon \times \text{Result}
\]
end
A tolerant style

\[
sqrt (x, \text{epsilon}: \text{REAL}): \text{REAL} \text{ is}
\]
\[
\text{-- Square root of } x, \text{ precision epsilon}
\]
require
\[
\text{True}
\]
do
\[
\text{if } x < 0 \text{ then}
\]
\[
\text{... Do something about it (?) ...}
\]
else
\[
\text{... normal square root computation ...}
\]
\[
\text{computed := True}
\]
end
ensure
\[
\text{computed \textit{implies}}
\]
\[
\text{abs (Result} ^ 2 - x) \leq 2 * \text{epsilon} * \text{Result}
\]
end
Contrasting styles

\[
\text{\textbf{put} (x: G) is} \quad \begin{cases} \text{Push } x \text{ on top of stack.} \\ \text{require} \quad \text{not } \text{is\_full} \\ \text{do} \quad \ldots \quad \end{cases}
\]

\[
\text{\textbf{tolerant\_put} (x: G) is} \quad \begin{cases} \text{Push } x \text{ if possible, otherwise set } \text{impossible} \text{ to } \text{True.} \\ \text{do} \quad \begin{cases} \text{if not } \text{is\_full} \text{ then} \\ \quad \text{put} (x) \\ \text{else} \\ \qquad \text{impossible} := \text{True} \\ \text{end} \quad \end{cases} \quad \end{cases} \end{cases}
\]
Invariants and “business rules”

- Invariants are absolute consistency conditions.
- They can serve to represent business rules if knowledge is to be built into the software.

Form 1

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

Form 2

```
invariant
not_under_minimum_if_normal:
  normal_state implies
  (balance >= Minimum_balance)
```
A powerful assertion language

- Assertion language:
  - Not first-order predicate calculus
  - But powerful through:
    - Function calls
  - Even allows to express:
    - Loop properties
Another contract mechanism

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

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

- Scheme 2 (guaranteeing without testing):

```plaintext
my_stack.remove
check
    my_stack_not_full: not my_stack.is_full
end
my_stack.put (some_element)
```
Design by Contract: technical benefits

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

- The need for exceptions arises when a 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.
The original strategy

\[ \begin{align*}
 r (\ldots) & \text{ is} \\
 & \text{require} \\
 & \ldots \\
 & \text{do} \\
 & \quad \text{op}_1 \\
 & \quad \text{op}_2 \\
 & \ldots \\
 & \quad \text{op}_i \\
 & \ldots \\
 & \quad \text{op}_n \\
 & \text{ensure} \\
 & \ldots \\
 & \text{end}
\end{align*} \]

Fails, triggering an exception in \( r \) (\( r \) is \textit{recipient} of exception).
Causes of exceptions

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

Handling exceptions properly

- Safe exception handling principle:
  - There are only two acceptable ways to react for the recipient of an exception:
    - Concede failure, and trigger an exception in the caller (Organized Panic)
    - Try again, using a different strategy (or repeating the same strategy) (Retry)}
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
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 is 100

attempt_transmission (message: STRING) is
  -- Transmit message in at most
  -- Max_attempts attempts.
  local
  failures: INTEGER
    do
      unsafe_transmit (message)
    rescue
      failures := failures + 1
      if failures < Max_attempts then
        retry
      end
    end
end
Transmitting over an unreliable line (2)

Max_attempts: INTEGER is 100
failed: BOOLEAN

attempt_transmission (message: STRING) is
  -- Try to transmit message;
  -- if impossible in at most Max_attempts
  -- attempts, set failed to true.
local
  failures: INTEGER
  do
    if failures < Max_attempts then
      unsafe_transmit (message)
    else
      failed := True
    end
  rescue
    failures := failures + 1
    retry
  end
If no exception clause (1)

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

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

- is an abbreviation for

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

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

- (1-n) For every exported routine \( r \):
  \[
  \{\text{INV and } \text{Pre}_r\} \text{ do}_r \{\text{Post}_r \text{ and INV}\}
  \]

- (1-m) For every creation procedure \( cp \):
  \[
  \{\text{Pre}_cp\} \text{ do}_cp \{\text{Post}_cp \text{ and INV}\}
  \]
Exception correctness

- For the normal body:

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

- For the exception clause:

\{ ??? \} \text{ rescue}_r \{ ??? \}
Exception correctness

- For the normal body:
  \[ \{\text{INV and Pre}_r\} \; \text{do}_r \; \{\text{Post}_r \; \text{and} \; \text{INV}\} \]

- For the exception clause:
  \[ \{\text{True}\} \; \text{rescue}_r \; \{\text{INV}\} \]
Absence of a rescue clause is equivalent to a default rescue clause:

\[ f (...) \text{ is} \]
\[ \begin{array}{l}
  \text{do} \\
  \quad \text{...} \\
  \text{end} \\
\end{array} \]

is an abbreviation for

\[ f (...) \text{ is} \]
\[ \begin{array}{l}
  \text{do} \\
  \quad \text{...} \\
  \text{rescue} \text{ default_rescue} \\
  \text{end} \\
\end{array} \]

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

- Use class `EXCEPTIONS` from the Kernel Library.

- Some features:
  - `exception` (code of last exception that was triggered).
  - `assertion_violation`, etc.
  - `raise` ("exception_name")
Design by Contract outside of Eiffel

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

- In programming languages:
  - Macros
  - Preprocessor

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

- Difficulties: contract inheritance; “short”-like tools; link with exception mechanism.
The possibility of direct assignments

\[ x.\text{attrib} = \text{value} \]

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

\[ x.\text{set\_attrib} (\text{value}) \]

with

\[
\text{set\_attrib} (v: \text{TYPE}) \textbf{is} \\
-- \text{Make } v \text{ the next value for } \text{attrib.} \\
\textbf{require} \\
... \text{Some condition on } v ... \\
\textbf{do} \\
\text{attrib} := v \\
\textbf{ensure} \\
\text{attrib} = v \\
\textbf{end}
\]
GNU Nana: improved support for contracts and logging in C and C++.

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

Support for time-related contracts (“Function must execute in less than 1000 cycles”).
Nana example

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

void 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); /* code must take less than 1000 cycles */
    DI(((r * r) <= $r) && ($r < (r + 1) * (r + 1))); /* use $r in postcondition */
}
```
In the basic package: no real notion of class invariant. ("Invariant", macro DI, is equivalent of "check" instruction.)

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

Macros: CHECK_NO, CHECK_REQUIRE, CHECK_ENSURE, CHECK_INVARIANT, CHECK_LOOP, CHECK_ALL.

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

No support for contract inheritance.
Java

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

- “Assert” instruction recently added.

- Gosling (May 1999):
  - “The number one thing people have been asking for is an assertion mechanism. Of course, that [request] is all over the map: There are people who just want a compile-time switch. There are people who … want something that's more analyzable. Then there are people who want a full-blown Eiffel kind of thing. We're probably going to start up a study group on the Java platform community process.”

iContract


- 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.
iContract example

```java
/**
 * @invariant age_ > 0
 */

class Person {

    protected age_;  

    /**
     * @post return > 0
     */
    int getAge() {...}

    /**
     * @pre age > 0
     */
    void setAge( int age ){...}

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

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

**OCL*-like assertion elements**
- `forall Type t in <enumeration> | <expr>`
- `exists Type t in <enumeration> | <expr>`
- `<a> implies <b>`

(* OCL: Object Constraint Language)
Another Java tool: Jass (Java)

- Preprocessor. Also adds Eiffel-like exception handling.
  
  http://theoretica.Informatik.Uni-Oldenburg.DE/~jass

```java
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{}; **/
}
```
- Adds assertions to Java, through modifications of the JDK 1.2 compiler.

- Cynthia della Torre Cicalessa

- See IEEE Computer, July 1999
The Object Constraint Language

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

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

- Not directly intended for execution.

- Jos Warmer, AW
OCL examples

- **Postconditions:**
  
  post: result = collection→iterate
  (elem; acc : Integer = 0 | acc + 1)

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

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

- Collection types include **Collection, Set, Bag, Sequence.**
Contracts for COM and Corba


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