Software architecture
Bertrand Meyer

*Design by Contract™*

**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

**Design by Contract**
- A discipline of analysis, design, implementation, management

**The French Driver’s License issue**

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

**Design by Contract**
- Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users).
- This goal is the element’s contract.
- The contract of any software element should be
  - Explicit.
  - Part of the software element itself.
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!

Ariane-5 (Continued)

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

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).

A human contract

<table>
<thead>
<tr>
<th>deliver</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</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>Supplier</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>

Contracts for analysis

```
deferred class PLANE

feature
    start_take_off is -- Initiate take-off procedures.
        require
            controls.passed 
            assigned_runway.is_clear
        deferred
            assigned_runway.owner = Current
            moving
        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
```

Class Invariant

- i.e. specified only.
- not implemented.

Precondition

Postcondition
Contracts for analysis

deferred class
TANK
feature
in_valve, out_valve: VALVE
{0}
require
-- Fill the tank.
in_valve.open
out_valve.is_closed
deferred
-- i.e. specified only.
in_valve.is_closed
out_valve.is_closed
ensure
-- not implemented.
in_valve.is_closed
out_valve.is_closed
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>Client</td>
<td>(Satisfy precondition:) Make sure input valve is open, output valve is closed.</td>
<td>(From postcondition:) Get filled-up tank, with both valves closed.</td>
</tr>
<tr>
<td>Supplier</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>

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 pre-condition (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

\( x >= 0 \)

... Square root algorithm to compute \( y \) ...

\( \{ abs \ (y ^ 2 - x) <= 2 * epsilon * y \} \)
  -- i.e.: \( y \) approximates exact square root of \( x \)
  -- within \( epsilon \)

So, is it like “assert.h”?

(Source: Reto Kramer)

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

```eiffel
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
```

A class without contracts

```eiffel
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
end
```

A contract (from EiffelBase)

```eiffel
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
```

A class without contracts

```eiffel
feature -- Deposit and withdrawal operations
  deposit (sum: INTEGER) is
    -- Deposit sum into the account.
    do
      add (sum)
    end
  withdraw (sum: INTEGER) is
    -- Withdraw sum from the account.
    do
      add (-sum)
    end
  may_withdraw (sum: INTEGER): BOOLEAN is
    -- Is it permitted to withdraw sum from the account?
    do
      Result := (balance - sum >= Minimum_balance)
    end
end
```

The contract

<table>
<thead>
<tr>
<th>Routine</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>PRECONDITION</td>
<td>POSTCONDITION</td>
</tr>
<tr>
<td>Supplier</td>
<td>POSTCONDITION</td>
<td>PRECONDITION</td>
</tr>
</tbody>
</table>

Introducing contracts

```eiffel
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`
  - `do` -- Add `sum` to the balance (secret procedure).
  - `ensure`
    - `increased: balance = old balance + sum`

**feature** -- Deposit and withdrawal operations
- `deposit (sum: INTEGER) is`
  - `require`
    - `not_too_small: sum >= 0`
  - `do add (sum)`
  - `ensure`
    - `increased: balance = old balance + sum`

- `withdraw (sum: INTEGER) is`
  - `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`

---

### The contract

<table>
<thead>
<tr>
<th>withdraw</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>(Satisfy precondition:) Make sure <code>sum</code> is neither too small nor too big.</td>
<td>(From postcondition:) Get account updated with <code>sum</code> withdrawn.</td>
</tr>
<tr>
<td>Supplier</td>
<td>(Satisfy postcondition:) Update account for withdrawal of <code>sum</code>.</td>
<td>(From precondition:) Simpler processing: may assume <code>sum</code> 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><code>balance := balance + sum</code></td>
<td><code>balance = old balance + sum</code></td>
</tr>
</tbody>
</table>

### Introducing contracts

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

**invariant**
- `not_under_minimum: balance >= Minimum_balance`
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 $cp$:

  $\{\text{Pre}_c\} \text{ do } \{\text{Post}_c \text{ and INV}\}$

- For every exported routine $r$:

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

- The worst possible erroneous run-time situation in object-oriented software development:
  - Producing an object that does not satisfy the invariant of its own class.

Uniform Access

$\:
\begin{align*}
\text{balance} &= \text{deposits.total} - \text{withdrawals.total} \\
\end{align*}
\$

New version

- Feature $\langle$NONE$\rangle$ -- Deposit and withdrawal operations

  $\text{deposit (sum: INTEGER) is}$

  $\text{require}$

  $\text{not too small: sum } \geq 0$

  $\text{do}$

  $\text{add (sum)}$

  $\text{ensure}$

  $\text{increased: balance } = \text{old balance} + \text{sum}$

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

$\end{align*}$
**New version**

```eiffel
withdraw (sum: INTEGER) is
  -- Withdraw sum from the account.
  require
  not_too_small: sum >= 0
  not_too_big: sum <= balance - Minimum_balance
  do
    add (- sum)
    withdrawals.extend (create (WITHDRAWAL).make (sum))
  ensure
    decreased: balance = old balance - sum
    one_more: withdrawals.count = old withdrawals.count + 1
  end
```

**Initial version**

```eiffel
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
```

**New version**

```eiffel
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
```

**Correct version**

```eiffel
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
    deposit (initial_amount)
  ensure
    balance_set: balance = initial_amount
  end
```

**The correctness of a class**

- For every creation procedure `cp`:
  `{Pre_p} do {Post_p and INV}
- For every exported routine `r`:
  `{INV and Pre_r} do {Post_r and INV}

**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. if ... then ... else...).
- A run-time assertion violation is something else: the manifestation of A DEFECT ("BUG")

The imperative and the applicative

<table>
<thead>
<tr>
<th>Prescriptive</th>
<th>Descriptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance := balance - sum</td>
<td>balance = old balance - sum</td>
</tr>
</tbody>
</table>

- How? | What? |
- Operational | Denotational |
- Implementation | Specification |
- Command | Query |
- Instruction | Expression |
- Imperative | Applicative |

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):

- 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):

- For special cases (e.g. "if the sum is negative, report an error...") use standard control structures (e.g. if ... then ... else...).
- A run-time assertion violation is something else: the manifestation of A DEFECT ("BUG")

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.
Contracts and quality assurance

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

\{P\} A \{Q\}

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 bug types

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

```
YOUR APPLICATION

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

```
COMPONENT LIBRARY

class LIST [G]

insert(x: G; i: INTEGER) is

require

j >= 0

j <= count + 1
```

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

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

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:

```java
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
```

Class example (continued)

```java
withdraw (sum: INTEGER) is -- Withdraw sum from the account.
require not_too_small: sum >= 0
not_too_big: sum <= balance - Minimum_balance
do  add (- sum)
withdrawals.extend (create (WITHDRAWAL), make (sum))
ensure decreased: balance = old balance - sum
one_more: withdrawals.count = old withdrawals.count + 1
end
```

Class example (continued)

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

Class example (end)

```java
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
```

Class example (continued)

```java
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
```

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
  ```
  feature (A, B, C)
  # (...) is
  require
  some_property
  ```

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

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.

Contract form of ACCOUNT class

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

withdraw (sum: INTEGER)  
  require
  not_too_small: sum >= 0
  not_too_big: sum <= balance - Minimum_balance
  ensure
  decreased: balance = old balance - sum
  one_more: withdrawals_count = old withdrawals_count + 1
  may_withdraw (sum: INTEGER): BOOLEAN
    -- Is it permitted to withdraw sum from the account?

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

Uses of the contract and interface forms

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

Contracts and reuse

- The contract form — i.e. the set of contracts governing a class — should be the standard form of library documentation.
- Reuse without a contract is sheer folly.
- See the Ariane 5 example.
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.

Assertion redeclaration rule

- When redeclaring a routine:
  - Precondition may only be kept or weakened.
  - Postcondition may only be kept or strengthened.
  - Redeclarion covers both redefinition and effecting.
  - Should this remain a purely methodological rule? A compiler can hardly infer e.g. that:
    \[ n > 1 \text{ implies (is stronger than) } n^{26} + 3 \times n^{25} > 3 \]

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
    ```

Contracts and inheritance

Correct call:
```eiffel
if a1.a then
  a1.r(...) -- Here a1.b holds.
end
```
Sequential structures

\begin{itemize}
  \item \textit{count} \textit{index} \textit{back} \textit{forth} before after
  \item \textit{start} \textit{index} \textit{back} \textit{forth} count
\end{itemize}

Sequential structures (continued)

\[\text{index} = \text{old index} + 1\]

\textbf{forth is}
\begin{itemize}
  \item require \textit{not after}
  \item deferred
  \item ensure \textit{index} \textit{after} \textit{before}
\end{itemize}

\textbf{start is}
\begin{itemize}
  \item deferred
  \item ensure \textit{empty or else} \textit{index} \textit{after} \textit{before}
\end{itemize}

Implementation variants

\begin{table}
\begin{tabular}{|c|c|c|c|c|}
\hline
start & forth & after & found (x) \\
\hline
\textbf{Arrayed list} & i := 1 & i := i + 1 & i > count & f [i] = x \\
\hline
\textbf{Linked list} & c := first cell & c := c.right & c := Void & c.item = x \\
\hline
\textbf{File} & rewind & read & end of file & f = x \\
\hline
\end{tabular}
\end{table}

Methodological notes

- Contracts are not input checking tests...
- ...but they can be used to help weed out undesirable input.
- Filter modules:
\begin{itemize}
  \item External objects
  \item Input and validation modules
  \item Processing modules
\end{itemize}
### 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)
```

### Not defensive programming!

- 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...
        else...
        end
    ensure
        abs (Result^2 - x) <= 2 * epsilon * Result
end
```

### Another example

```plaintext
sqrt (x, epsilon: REAL): REAL is
    -- Square root of x, precision epsilon
    require
        x >= 0
        epsilon >= 0
    do ...
    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.

### The contract

<table>
<thead>
<tr>
<th></th>
<th>OBLIGATIONS</th>
<th>BENEFITS</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>

### Interpreters

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

A demanding style

\[
\text{sqrt} \ (x, \epsilon) : \text{REAL} \rightarrow \text{REAL} \quad \text{is} \\
\quad \text{-- Square root of } x, \text{ precision } \epsilon \quad \text{-- Same version as before} \\
\text{require} \\
\quad \begin{align*}
& x \geq 0 \\
& \epsilon \geq 0
\end{align*} \\
\text{do} \\
\quad \ldots \\
\text{ensure} \\
\quad \begin{align*}
& \text{abs} \ (\text{Result}^2 - x) \leq 2 \times \epsilon \times \text{Result} \\
\end{align*} \\
\text{end}
\]

A tolerant style

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

Contrasting styles

- \[ \text{put} \ (x: G) \quad \text{is} \quad \text{-- Push } x \text{ on top of stack.} \]
- \[ \text{require} \\
\quad \begin{align*}
& \text{not is_full} \\
\end{align*} \\
\text{do} \\
\quad \ldots \\
\text{end} \]
- \[ \text{tolerant_put} \ (x: G) \quad \text{is} \quad \text{-- Push } x \text{ if possible, otherwise set impossible to } \text{True}. \]
- \[ \text{do} \\
\quad \begin{align*}
& \text{if not } \text{is_full} \text{ then} \\
& \quad \text{put} \ (x) \\
& \text{else} \\
& \quad \text{impossible} := \text{True} \\
\end{align*} \\
\text{end} \]

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
  \[ \text{invariant} \\
  \quad \begin{align*}
  & \text{not_under_minimum}: \text{balance} \geq \text{Minimum_balance} \\
  \end{align*} \]
- Form 2
  \[ \text{invariant} \\
  \quad \begin{align*}
  & \text{not_under_minimum_if_normal}: \\
  & \quad \text{normal_state implies} \\
  & \quad \begin{align*}
  & \text{balance} \geq \text{Minimum_balance} \\
  \end{align*} \\
  \end{align*} \]

A powerful assertion language

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

- 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.

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.

Precondition design

- Scheme 2 (guaranteeing without testing):
  ```
  my_stack.remove
  check
      my_stack_not_full: not my_stack.is_full
  end
  my_stack.put (some_element)
  ```

Exception handling

- The need for exceptions arises when the contract is broken.
- Two concepts:
  - Failure: a routine, or other operation, is unable to fulfill its contract.
  - Exception: an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.

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.

The original strategy

```
r (...) is require
  ... do
      op;
      op;
      ... op
      ... op;
  ensure
  end
```

Fails, triggering an exception in `r` (r is 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) (Retrying).

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");
  when others => ... 
end; -- sqrt
```

Transmitting over an unreliable line (1)

```ada
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
```
### Transmitting over an unreliable line (2)

```plaintext
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
  retry
rescue
  failures := failures + 1
end
```

### Exception correctness: A quiz

- For the normal body:

  `{INV and prer} do. {INV and postr}`

- For the exception clause:

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

### If no exception clause (1)

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

  ```plaintext
  f(...) is
d... end
  ```
  is an abbreviation for

  ```plaintext
  f(...) is
d... rescue
  ```
  -- Nothing here

- (This is a provisional rule; see next.)

### Quiz answers

- For the normal body:

  `{INV and prer} do. {INV and postr}`

- For the exception clause:

  `{True} rescue. {INV}`

### The correctness of a class

- (1-n) For every exported routine `r`:

  `{INV and prer} do. {INV and postr}`

- (1-m) For every creation procedure `cp`:

  `{prea} doa. {posta and INV}`

### Bank accounts

```plaintext
balance := deposits.total - withdrawals.total
```

(A2)
If no exception clause (2)

- Absence of a rescue clause is equivalent to a default rescue clause:
  
  ```
  f (...) do
  end
  ```

  is an abbreviation for
  
  ```
  f (...) do
  rescue default_rescue
  end
  ```

- The task of `default_rescue` is to restore the invariant.

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.

For finer-grain exception handling

- Use class `EXCEPTIONS` from the Kernel Library.

- Some features:
  - `exception` (code of last exception that was triggered).
  - `is_assertion_violation`, etc.
  - `raise` ("exception_name")

C++/Java Design by Contract limitations

- The possibility of direct assignments

  ```
  x.attrib = value
  ```

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

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

  with

  ```
  set_attrib (v: TYPE) is
  require ... Some condition on v ...
  do attrib := v ensure attrib = v end
  ```

C++ Contracts

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

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

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

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){...}
    ...
}
```

iContract specification language

- Any expression that may appear in an if(...) condition may appear in a precondition, postcondition or invariant.
- Scope:
  - Invariant: as if it were a routine of the class.
  - Precondition and postcondition: as if they were part of the routine.
- OCL*-like assertion elements
  - forall Type t in <enumeration> | <expr>
  - exists Type t in <enumeration> | <expr>
  - <a> implies <b>
  - (* OCL: Object Constraint Language)
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{}; **/
}
```

Biscotti

- Adds assertions to Java, through modifications of the JDK 1.2 compiler.
- Cynthia della Torre Cicalese
- 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