Lecture 4: Design by Contract

Reading material

OOSC2:
> Chapter II: Design by Contract

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!

With assertions, the error would almost certainly (if not avoided in the first place) detected by either static inspection or testing:

```
integer_bias (b: REAL): INTEGER is
  require representable (b)
  do ...
  ensure equivalent (b, Result)
end
```

Ariane 5 (Conclusion)
The main lesson:
Reuse without a contract is sheer folly

See:
Jean-Marc Jézéquel and Bertrand Meyer
Design by Contract: The Lessons of Ariane
IEEE Computer, January 1997
Also at http://www.eiffel.com

A human contract

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

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.

Contracts for analysis, specification

Contracts for analysis

<table>
<thead>
<tr>
<th></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.

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 \( y \) ...

\[ \begin{align*}
& \text{abs} (y^2 - x) \leq 2 \times \epsilon \\
& \text{-- i.e.: } \text{y approximates exact square root of } x \\
& \text{-- within } \epsilon
\end{align*} \]

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. \( \{ \text{False} \} A \{ \ldots \} \)
2. \( \{ \ldots \} A \{ \text{True} \} \)

A contract (from EiffelBase)

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

-- Assuming there is no item of key key, 
-- insert new with key; set inserted.

\[ \text{require} \]

key_not_present: \( \text{not has (key)} \)

\[ \text{ensure} \]

insertion_done: \( \text{item (key) = new} \)

key_present: \( \text{has (key)} \)

inserted: \( \text{inserted} \)

one_more: \( \text{count = old count + 1} \)

The contract

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

A class without contracts

\[ \text{class ACCOUNT} \]

\[ \text{feature -- Access} \]

\[ \text{balance: INTEGER} \]

-- Balance

\[ \text{Minimum_balance: INTEGER is 1000} \]

-- Minimum balance

\[ \text{feature (NONE) -- Deposit and withdrawal} \]

\[ \text{add (sum: INTEGER) is} \]

-- Add \( \text{sum} \) to the \( \text{balance} \) (secret procedure).

\[ \text{do} \]

\[ \text{balance := balance + sum} \]

\[ \text{end} \]

A class without contracts

\[ \text{feature -- Deposit and withdrawal operations} \]

\[ \text{deposit (sum: INTEGER) is} \]

-- Deposit \( \text{sum} \) into the account.

\[ \text{do} \]

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

\[ \text{end} \]

\[ \text{withdraw (sum: INTEGER) is} \]

-- Withdraw \( \text{sum} \) from the account.

\[ \text{do} \]

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

\[ \text{end} \]

\[ \text{may_withdraw (sum: INTEGER, BOOLEAN) is} \]

-- Is it permitted to withdraw \( \text{sum} \) from the account?

\[ \text{do} \]

\[ \text{Result := (balance - sum >= Minimum_balance)} \]

\[ \text{end} \]
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

```plaintext
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
```

Introducing contracts

```plaintext
feature -- Deposit and withdrawal operations
deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  require
    not_too_small: sum >= 0
    add (sum)
  ensure
    increased: balance = old balance + sum
end
```

The contract

<table>
<thead>
<tr>
<th>withdrew (sum: INTEGER) is</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- Withdraw sum from the account.</td>
</tr>
<tr>
<td>require</td>
</tr>
<tr>
<td>not-too-small: sum &gt;= 0</td>
</tr>
<tr>
<td>not-too-big: sum &lt;= balance - Minimum_balance</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>add (-sum)</td>
</tr>
<tr>
<td>-- i.e. balance := balance - sum</td>
</tr>
<tr>
<td>ensure</td>
</tr>
<tr>
<td>decreased: balance = old balance - sum</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

The imperative and the applicative

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

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

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

The class invariant

Consistency constraint applicable to all instances of a class.

Must be satisfied:
\[ \checkmark \text{After creation.} \]
\[ \checkmark \text{After execution of any feature by any client.} \]
(Qualified calls only: \(x.f(\ldots)\))

The correctness of a class

For every creation procedure \(cp\): \[\{\text{Pre}_{cp}\} \text{ do } \{\text{INV and Post}_{cp}\}\]
For every exported routine \(r\): \[\{\text{INV and Pre}_{r}\} \text{ do } \{\text{INV and Post}_{r}\}\]

Uniform Access

\text{balance = deposits.total - withdrawals.total}

A slightly more sophisticated version

\text{class ACCOUNT}
\text{create make}

\text{feature (NONE) -- Implementation}
\text{add (sum : INTEGER) is}
\text{-- Add sum to the balance (secret procedure).}
\text{do}
\text{balance := balance + sum}
\text{ensure balance\_increased: balance = old balance + sum}
\text{end}

\text{deposits: DEPOSIT\_LIST}
\text{withdrawals: WITHDRAWAL\_LIST}

New version

\text{feature (NONE) -- Initialization}
\text{make (initial\_amount: INTEGER) is}
\text{-- Set up account with initial\_amount}
\text{require}
\text{do}
\text{large\_enough: initial\_amount \geq \text{Minimum\_balance}}
\text{balance := initial\_amount}
\text{create deposits, make}
\text{create withdrawals, make}
\text{ensure balance\_set: balance = initial\_amount}
\text{end}

\text{feature -- Access}
\text{balance: INTEGER}
\text{-- balance}
\text{Minimum\_balance: INTEGER is 1000}
\text{-- Minimum balance}
New version

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
    one_more: deposits.count = old deposits.count + 1
end

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

may_withdraw (sum: INTEGER, BOOLEAN) is
  -- 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

create x.make (...)  37

For every creation procedure cp:

{Pre cp} do cp {INV and Post cp}

For every exported routine r:

{INV and Pre r} do r {INV and Post r}

Initial 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

Correct 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
    deposit (initial_amount)
  ensure balance_set: balance = initial_amount
end

The page contains a class definition for a bank account system. The class includes features for deposit and withdrawal operations, as well as a method to determine if it is permitted to withdraw a certain amount. The correctness of the class is verified through preconditions, postconditions, and invariants. The updated version includes more detailed checks and ensures that the account balance is maintained correctly.
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: run-time effect

Compilation options (per class, in Eiffel):
- No assertion checking
- Preconditions only
- Preconditions and postconditions
- Preconditions, postconditions, class invariants
- All assertions

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):

```plaintext
put (x: G) is
   require not is_full
   do ...
   ensure not is_empty
end
```

```
remove is
   require not is_empty
   do ...
   ensure not is_full
end
```

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></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESCRIPTIVE</td>
<td>DESCRIPTIVE</td>
</tr>
<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>
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”)

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 and bug types

Preconditions are particularly useful to find bugs in client code:

```
class LIST [G]
  insert (x: G; i: INTEGER) is
    require
      i >= 0
      i <= count + 1
    end insert;
```

YOUR APPLICATION

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

COMPONENT LIBRARY

```
insert (x: G; i: INTEGER)
```

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

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.
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.

Class example (continued)

```plaintext
feature -- Deposit and withdrawal operations
deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  require
  not_too_small: sum >= 0
  do
deposit (sum) 
deposits.extend (create (DEPOSIT).make (sum))
  ensure
  increased: balance = old balance + sum
end
```

Class example (continued)

```
withdraw (sum: INTEGER) is
  -- Withdraw sum from the account.
  require
  not_too_small: sum >= 0
  not_too_big: sum <= balance - Minimum_balance
  do
withdraw (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)

```plaintext
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
```

Class example (continued)

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

Class example (end)

```
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}
  f(...) is
  require
  some_property
```

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

Contract form of ACCOUNT class

```plaintext
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
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?

Contracts and inheritance

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.

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

\[ n^{25} + 3 \times n^{25} > 3 \]

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

\[
\begin{align*}
\text{require else new_pre}\\
\text{ensure then new_post}
\end{align*}
\]

Resulting assertions are:
  > new_pre or else original_precondition
  > original_postcondition and then new_post

Assertion redeclaration rule in Eiffel
Don't call us, we'll call you

Deferred class `LIST[G]`
inherit
`CHAIN[G]`
feature
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 (continued)

index: INTEGER is
defered
end
... empty, found, after, ...

invariant
0 <= index
index <= size + 1
empty implies (after or before)
end

Sequential structures

Sequential structures (continued)

Descendant implementations

Implementation variants
Methodological notes

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

Filter modules:

Precondition design

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

Scheme 1 (testing):

\[
\begin{align*}
&\text{if not my_stack.is_full then} \\
&\quad \text{my_stack.put (some_element)} \\
&\text{end}
\end{align*}
\]

Scheme 2 (guaranteeing without testing):

\[
\begin{align*}
&\text{my_stack.remove} \\
&\text{... my_stack.put (some_element)}
\end{align*}
\]

The contract

<table>
<thead>
<tr>
<th>Client</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfy precondition:</td>
<td>Provide non-negative value and precision that is not too small.</td>
<td>Get square root within requested precision.</td>
</tr>
<tr>
<td>Supplier</td>
<td>Satisfy postcondition:</td>
<td>Produce square root within requested precision.</td>
</tr>
<tr>
<td>Simplier processing thanks to assumptions on value and precision.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another example

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

Not defensive programming!

It is never acceptable to have a routine of the form

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

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.
Interpreters

```plaintext
class BYTExCOODE_PROGRAM
feature
  verified BOOLEAN
trustful_execute (program BYTExCOODE) is
  require ok verified
do...
end
end distrufful_execute (program BYTExCOODE) is
do verify
if verified then
  trustful_execute (program)
end if is do
end end verify
```

A tolerant style

```plaintext
sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
require True
if x < 0 then
  ... Do something about it (?) ...
else
  ... normal square root computation ...
computed := True
end if verified then
trustful_execute (program)
end verify
ensure computed implies
  abs (Result^2 - x) <= 2 * epsilon * Result
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!

Contrasting styles

```plaintext
put (x: i) is
  -- Push x on top of stack.
require not is_full
do...
end
end
tolerant_put (x: i) is
  -- Push x if possible, otherwise set impossible to True.
do if not is_full then
  put (x)
else
  impossible := True
end
```

A demanding style

```plaintext
sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  -- Same version as before
require
  x >= 0 epsilon
do...
ensure
  abs (Result^2 - x) <= 2 * epsilon * Result
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

```plaintext
invariant not_under_minimum: balance <> Minimum_balance
```

Form 2

```plaintext
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

### 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.

### 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.

### 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):

```plaintext
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.
Analysis classes

deferred class VAT
  inherit TANK
  feature
    in_valve, out_valve: VALVE
    fill
      require in_valve.open
      out_valve.closed
    deferred
      in_valve.closed
      out_valve.closed
    end
    is_full
      empty, is_full, is_empty, gauge, maximum, ...
    invariant
      is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)
  end

What is object-oriented analysis?

- Classes around object types (not just physical objects but also important concepts of the application domain)
- Abstract Data Types approach
- Deferred classes and features
- Inter-component relations: "client" and inheritance
- Distinction between reference and expanded clients
- Inheritance — single, multiple and repeated for classification.
- Contracts to capture the semantics of systems: properties other than structural.
- Libraries of reusable classes

Why O-O analysis?

Same benefits as O-O programming, in particular extendibility and reusability

Direct modeling of the problem domain

Seamlessness and reversibility with the continuation of the project (design, implementation, maintenance)

What O-O requirements analysis is not

Use cases

(Not appropriate as requirements statement mechanism)

Use cases are to requirements what tests are to specification and design

Television station example

class SCHEDULE
  feature
    segments: LIST [SEGMENT]
  end

Schedules

deferred class SCHEDULE
  feature
    segments: LIST [SEGMENT]
    -- Successive segments
    deferred
    end
    air_time: DATE
    -- 24-hour period for this schedule
    deferred
    end
    set_air_time (t: DATE)
    -- Assign schedule to be broadcast at time t.
    require t.in_future
    deferred
    ensure air_time = t
    end
    print
    -- Produce paper version
    deferred
    end
  end

Source: OOSC
Contracts

Feature precondition: condition imposed on the rest of the world

Feature postcondition: condition guaranteed to the rest of the world

Class invariant: Consistency constraint maintained throughout on all instances of the class

Obligations & benefits in a 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>

Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:

> Committing too early to an implementation
  Overspecification

> Missing parts of the problem
  Underspecification)

Segment

Obligations & benefits in a contract (continued)

<table>
<thead>
<tr>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_list: (I &lt;= index) and (index &lt;= schedule.segments.count)</td>
</tr>
<tr>
<td>in_schedule: schedule.segments.item (index) = Current</td>
</tr>
<tr>
<td>next_in_list: (next /= Void) implies (schedule.segments.item (index + 1) = next)</td>
</tr>
<tr>
<td>no_next_iff_last: (next = Void) = (index = schedule.segments.count)</td>
</tr>
<tr>
<td>positive_times: (starting_time &gt; 0) and (ending_time &gt; 0)</td>
</tr>
<tr>
<td>sufficient_duration: ending_time – starting_time &gt;= Minimum_duration</td>
</tr>
<tr>
<td>decent_interval: (next.starting_time) - ending_time &lt;= Maximum_interval</td>
</tr>
</tbody>
</table>

Commercial

Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:

> Committing too early to an implementation
  Overspecification

> Missing parts of the problem
  Underspecification)
Diagrams: UML, BON

Text-Graphics
Equivalence

Causes of exceptions

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

Routine call

Transmitting over an unreliable line (1)

```
Max_attempts: INTEGER is 100

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
      unsafe_transmit(message)
      if failures < Max_attempts then
        retry
      end
    end
  rescue
    failures := failures + 1
    retry
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
    end
  rescue
    failures := failures + 1
    retry
end
```

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).

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
  rescue
    Nothing here
  end
```

(This is a provisional rule; see next.)

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:

(precp) do (postcp and INV)
Exception correctness: A quiz

For the normal body:

\{(INV and pre) do (INV and post)\}

For the exception clause:

\{(???) rescue (???)\}

Quiz answers

For the normal body:

\{(INV and pre) do (INV and post)\}

For the exception clause:

\{(True) rescue (INV)\}

Bank accounts

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

(A2)

If no exception clause (2)

Absence of a rescue clause is equivalent to a default rescue clause:

\[f(...) is do ~ end\]

is an abbreviation for:

\[f(...) is do ~ rescue default_rescue end\]

The task of \textit{default_rescue} is to restore the invariant.

Exception handling

Use class \textit{EXCEPTIONS} from the Kernel Library.

Some features:

- \textit{exception} (code of last exception that was triggered).
- \textit{is_assertion_violation}, etc.
- \textit{raise} \textit{exception_name}.

Agenda for today

Exception handling

Design by Contract outside of Eiffel
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.

C++/Java Design by Contract limitations

The possibility of direct assignments

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

```c++
x.attrib = value
```

with

```
set_attrib (v: TYPE)
```

- require: Make v the next value for attrib.
- do: Some condition on v ...
- ensure: attrib = v
- end: attrib = v

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").

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."

(http://www.javaworld.com/javaworld/jaw01-99-gosling.html)
**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.

**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(); **/ 
}
```

**iContract example**

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

class Person {
    protected
    /**
     * @post return > 0
     */
    int getAge() {...}
    /**
     * @pre age > 0
     */
    void setAge(int age){...}
    ...
}
```

**Biscotti**

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

Cynthia della Torre Cicalése

See *IEEE Computer, July 1999*

**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>`
- `<o implies <b>`

(* OCL: Object Constraint Language*

**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:

\[
\text{post: result = collection->iterate} \\
\quad \text{if elem = object then acc + 1 else acc endif)
\]

\[
\text{post: result = collection->iterate} \\
\quad \text{if elem = object then acc + 1 else acc endif)
\]

\[
\text{post: T.allInstances->forAll} \\
\quad \text{if elem = object then acc + 1 else acc endif)
\]

Collection types include Collection, Set, Bag, Sequence.

Contracts for COM and Corba


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

Complementary material

OOSC2:

- Chapter 11: Design by Contract