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.

Design by Contract: applications

- Built-in correctness
- Automatic documentation
- Self-debugging, self-testing code
- Get inheritance right
- Get exceptions right
- Give managers better control tools
Design by Contract

Software construction: the underlying view

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

EiffelStudio documentation

Produced automatically from class text
Available in text, HTML, Postscript, RTF, FrameMaker and many other formats
Numerous views, textual and graphical

Contracts for documentation

LINKED_LIST Documentation, generated by EiffelStudio

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. (What's the problem?)
The different forms of a class

<table>
<thead>
<tr>
<th>Elements from the class only</th>
<th>Elements from the class and its ancestors</th>
</tr>
</thead>
<tbody>
<tr>
<td>All features and contract clauses</td>
<td>Text view: text form Contract view: short form</td>
</tr>
<tr>
<td>Only the exported (non secret) elements</td>
<td>Flat view: flat form Interface view: flat short form</td>
</tr>
</tbody>
</table>

Export rule for preconditions

In

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.

Contracts for analysis, specification

defered class VAT inherit TANK

feature

in_valve, out_valve: VALVE

fill is

-- Fill the vat.
require
in_valve.open
out_valve.closed
defered
in_valve.closed
out_valve.closed
ensure

is_full

end

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

invariant

is_full \(= \) (gauge \(= \) 0.97 \(\times\) maximum) and (gauge \(= \) 1.03 \(\times\) maximum)

end
Contracts for testing and debugging

- Contracts express implicit assumptions behind code
- A bug is a discrepancy between intent and code
- Contracts state the intent!

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

May disable monitoring when releasing software
A revolutionary form of quality assurance

Lists in EiffelBase

Trying to insert too far right

Cursor (Already past last element!)
A command and its contract

Moving the cursor forward

Two queries, and command "forth"
Where the cursor may go

Valid cursor positions

From the invariant of class LIST

Valid cursor positions

Contract monitoring

A contract violation always signals a bug:

- Precondition violation: bug in client
- Postcondition violation: bug in routine
Contracts and inheritance

Issues: what happens, under inheritance, to

- Class invariants?
- Routine preconditions and postconditions?

Invariants

Invariant Inheritance rule:

- The invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed.

Accumulated result visible in flat and interface forms.

Contracts and inheritance

Correct call in C:
if a1.\alpha \text{ then}
  a1.r(...)
-- Here a1.\beta \text{ hold}
end

Correct call in C:
Assertion redeclaration rule

When redeclaring a routine, we may only:

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

Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

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

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

Resulting assertions are:

- `original_precondition` or `new_pre`
- `original_postcondition` and `new_post`

Contracts as a management tool

High-level view of modules for the manager:

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

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

Precondition design

The client must guarantee the precondition before the call.
This does not necessarily mean testing for the precondition.
Scheme 1 (testing):
\[
\begin{align*}
\text{if not } & \text{my\_stack\_is\_full} \\
& \text{my\_stack\_put(some\_element)} \\
\end{align*}
\]
Scheme 2 (guaranteeing without testing):
\[
\begin{align*}
\text{my\_stack\_remove} \\
\text{my\_stack\_put(some\_element)}
\end{align*}
\]

Another example

\[
sqrt(x, \text{epsilon: REAL}): \text{REAL}
\text{is}
\begin{align*}
\text{require} \\
& x \geq 0 \\
& \text{epsilon} \geq 0 \\
\text{do} \\
\text{ensure} \\
& \text{abs} (\text{Result}^2 - x) \leq 2 \text{ epsilon} \times \text{Result}
\end{align*}
\]
### The contract

<table>
<thead>
<tr>
<th>sqrt</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</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>Supplier</td>
<td>(Satisfy postcondition:) Produce square root within requested precision.</td>
<td>(From precondition:) Simpler processing thanks to assumptions on value and precision.</td>
</tr>
</tbody>
</table>

### Not defensive programming

It is not acceptable to have a routine of the form

```
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

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**

class BYTECODE_PROGRAM feature
  verified: BOOLEAN
  truthful_execute (program: BYTECODE) is
    verify ok verified
    do...
  end
  distrustful_execute (program: BYTECODE) is
    verify
    if verified then
      truthful_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**

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

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

**Contrasting styles**

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

tolerant_put (x: G) is
  -- Push x if possible, otherwise set impossible to
  -- True.
  do
    if not is_full then
      put (x)
    else
      impossible := True
    end
  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)
```
**Power of the assertion language**

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

**Loop trouble**

Loops can be hard to get right:
- "Off-by-one"
- Infinite loops
- Improper handling of borderline cases

For example: binary search feature

**The answer: loop contracts**

Use of loop variants and invariants.

A loop is a way to compute a certain result by successive approximations.

(e.g. computing the maximum value of an array of integers)
Computing the max of an array

\[ \text{highest}(\text{sl: LIST}[\text{STRING}]): \text{STRING is} \]
\[ \text{require} \]
\[ \text{sl/ Void} \]
\[ \text{not sl.is_empty} \]
\[ \text{do} \]
\[ \text{from sl.start; Result := ""} \]
\[ \text{until sl.after} \]
\[ \text{loop} \]
\[ \text{Result := greater (Result, sl.item)} \]
\[ \text{sl.forth} \]
\[ \text{end} \]

Loop as approximation strategy

Result = \( s_1 \)
Result = \( \text{Max} (s_1, s_2) \)
Result = \( \text{Max} (s_1, s_2, \ldots, s_i) \)
Result = \( \text{Max} (s_1, s_2, \ldots, s_i, \ldots, s_n) \)

The loop invariant

\[ \text{from sl.start; Result := ""} \]
\[ \text{invariant} \]
\[ \text{sl.index} = 1 \]
\[ \text{sl.index} = \text{sl.count} + 1 \]
\[ \text{-- Result is greatest of elements so far} \]
\[ \text{until sl.after} \]
\[ \text{loop} \]
\[ \text{Result := greater (Result, sl.item)} \]
\[ \text{sl.forth} \]
\[ \text{end} \]
Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied

The loop invariant

from
   sl.start ; Result := ""
 invariant
      sl.index >= 1
      sl.count <= sl.count + 1
      -- Result is greatest of elements so far
 until
 sl.after
 loop
   Result := greater (Result, sl.item)
 sl.forth
 end

The loop invariant (better)

from
   sl.start ; Result := ""
 invariant
      sl.index >= 1
      sl.count <= sl.count + 1
      -- If there are any previous elements, Result is the greatest
 until
 sl.after
 loop
   Result := greater (Result, sl.item)
 sl.forth
 end
The effect of the loop

from
sl.start ; Result := “”
invariant
sl.index >= 1
sl.index <= sl.count + 1
-- Result is greatest of elements so far
until
sl.after
loop
Result := greater (Result, sl.item)
sl.forth
end

Loop semantics rule

The effect of a loop is the combination of:

➢ Its invariant
➢ Its exit condition

How do we know the loop terminates?

from
sl.start ; Result := “”
invariant
sl.index >= 1
sl.index <= sl.count + 1
-- If there are any previous elements, Result is the greatest
until
sl.after
loop
Result := greater (Result, sl.item)
sl.forth
end
**Loop variant**

Integer expression that must:

- Be non-negative when after initialization (from)

- Decrease (i.e. by at least one), while remaining non-negative, for every iteration of the body (loop) executed with exit condition not satisfied

---

**The variant for our loop**

```plaintext
from
sl.start : Result := ""
variant
   sl.count - sl.index + 1
invariant
   sl.index >= 1
   sl.index <= sl.count + 1
   -- If there are any previous elements, Result is the greatest
until
   sl.after
loop
   Result := greater(Result, sl.item)
   sl.forth
end
```

---

**Another contract construct**

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):

\begin{verbatim}
  my_stack.remove
  check
    my_stack_not_full: not my_stack.is_full
  end
  my_stack.put (some_element)
\end{verbatim}