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
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.
Without contracts

feature -- Deposit and withdrawal operations

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

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

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

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
With contracts

**feature** -- Deposit and withdrawal operations

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

\[
\text{-- Deposit sum into the account.}
\]

\[
\text{require}
\]

\[
\text{not\_too\_small: sum} \geq 0
\]

\[
\text{do}
\]

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

\[
\text{ensure}
\]

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

\[
\text{end}
\]
**With contracts**

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

\[\text{-- Withdraw } \text{sum} \text{ from the account.}\]

**require**

\[\text{not\_too\_small: } \text{sum} \geq 0\]

\[\text{not\_too\_big:}\]

\[\text{sum} \leq \text{balance - Minimum\_balance}\]

**do**

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

\[\text{-- i.e. balance := balance - sum}\]

**ensure**

\[\text{decreased: } \text{balance} = \text{old balance} - \text{sum}\]

**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>PRESCRIPTOR</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>
With contracts

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

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

\[ \text{-- account?} \]

\[ \text{do} \]

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

\[ \text{end} \]

\[ \text{invariant} \]

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

\[ \text{end} \]
How not to do it

\[ r(i: \text{INTEGER}): \text{BOOLEAN} \] is
\begin{align*}
\text{require} & \quad i \geq 0 \text{ or } i < 0 \\
\text{ensure} & \quad \text{Result} = \text{true} \text{ or Result} = \text{false}
\end{align*}

\[ r(x: \text{BANK}\_\text{ACCOUNT}): \text{BOOLEAN} \] is
\begin{align*}
\text{require} & \quad x \neq \text{Void} \text{ and } x.\text{balance} > 0
\end{align*}
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: \( a.f (...) \))
Export rule for preconditions

In

\[ \text{feature } \{ A, B, C \} \]
\[ r (...) \text{ is} \]
\[ \text{require} \]
\[ \text{some\_property} \]

\text{some\_property} must be exported (at least) to \( A, B \) and \( C \)!
No such requirement for postconditions and invariants.
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:

\[
\text{if } a1.\alpha \text{ then}
\]

\[
a1.r(...) 
\]

-- Here a1.\beta holds

end
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

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

Resulting assertions are:

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

Is the example correct?

Inheritance
Example

```
withdraw (v: INTEGER)
require is_drawable (v)

is_drawable (v: INTEGER): BOOLEAN
do Result:= True end

is_drawable (v: INTEGER): BOOLEAN
do Result := balance > v + fee end

is_drawable (v: INTEGER): BOOLEAN
do Result := balance - v > credit end

Inheritance
```
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