Object-Oriented Software Construction

Bertrand Meyer
Lecture 9:
Design by Contract™
Design by Contract

- A discipline of analysis, design, implementation, management
Design by Contract (cont’d)

- 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.
## A human contract

<table>
<thead>
<tr>
<th></th>
<th><strong>OBLIGATIONS</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>Bring package before 4 p.m.; pay fee.</td>
<td>Get package delivered by 10 a.m. next day.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>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).
deferred class PLANE inherit AIRCRAFT

feature

start_take_off is

require controls.passed
assigned_runway.clear

defered 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
deferred class VAT inherit TANK

feature

  in_valve, out_valve: VALVE

fill is

  require
  in_valve.open
  out_valve.closed

  deferred ensure
  in_valve.closed
  out_valve.closed
  is_full

end

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

invariant

  is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)

end

Precondition

Postcondition

Class invariant

-- i.e. specified only.

-- not implemented.
Contracts for analysis (cont’d)

<table>
<thead>
<tr>
<th>fill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Satisfy precondition:) Make sure input valve is open, output valve is closed.</td>
<td>(Satisfy postcondition:) Get filled-up vat, with both valves closed.</td>
</tr>
<tr>
<td>(Satisfy postcondition:) Fill the vat 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>
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.
Some benefits: technical

- 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.
Some benefits: managerial

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

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

```
\{P\} A \{Q\}
```

“Hoare triple”

What this means (total correctness):

- Any execution of \(A\) started in a state satisfying \(P\) will terminate in a state satisfying \(Q\).
Hoare triples: a simple example

\{ n > 5 \} \ n := n + 9 \ \{ n > 13 \}

- Most interesting properties:
  - Strongest postcondition (from given precondition).
  - Weakest precondition (from given postcondition).

- "P is stronger than or equal to Q" means: P implies Q

- QUIZ: What is the strongest possible assertion? The weakest?
Specifying a square root routine

\{ x \geq 0 \}

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

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

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

-- within \( \text{epsilon} \)
Software correctness

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

- Take this as a job ad in the classifieds.

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

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

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
# 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>
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
Without contracts (cont’d)

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
Introducing contracts

**class** ACCOUNT **create**

*make*

**feature** \{NONE\} -- Initialization

*make* \( (\text{initial} \_\text{amount}:\;\text{INTEGER}) \text{ is} \)

-- Set up account with \text{initial} \_\text{amount}.

\begin{verbatim}
require
large\_enough: initial\_amount \geq Minimum\_balance

do
balance := initial\_amount

ensure
balance\_set: balance = initial\_amount
\end{verbatim}

*end*
Introducing contracts (cont’d)

feature -- Access

\textit{balance: INTEGER} \\
\quad -- Balance

\textit{Minimum\_balance: INTEGER} is 1000 \\
\quad -- Minimum balance

feature \{NONE\} -- Implementation of deposit and withdrawal

\textit{add (sum: INTEGER) is} \\
\quad -- Add \textit{sum} to the \textit{balance} (secret procedure).

\textbf{do} \\
\quad balance := balance + sum

\textbf{ensure} \\
\quad increased: balance = old balance + sum

\textbf{end}
feature -- Deposit and withdrawal operations

**deposit** (*sum*: INTEGER) is
-- Deposit *sum* into the account.

require

not_too_small: *sum* \( \geq 0 \)

do

add (*sum*)

ensure

increased: *balance* = old *balance* + *sum*

declare
With contracts (cont’d)

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

\[
\begin{align*}
\text{require} \quad & \not_{\text{too small}}: \quad \text{sum} \geq 0 \\
& \not_{\text{too big}}: \\
& \quad \text{sum} \leq \text{balance} - \text{Minimum\_balance}
\end{align*}
\]

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

\[
\begin{align*}
\text{ensure} \quad & \text{decreased: balance} = \text{old balance} - \text{sum}
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure <em>sum</em> is neither too small nor too big.</td>
<td>(From postcondition:) Get account updated with <em>sum</em> withdrawn.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Update account for withdrawal of <em>sum</em>.</td>
<td>(From precondition:) Simpler processing: may assume <em>sum</em> is within allowable bounds.</td>
</tr>
</tbody>
</table>
The imperative and the applicative

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

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

end
The class invariant

- Consistency constraint applicable to all instances of a class.

- Must be satisfied:
  - After creation.
  - After execution of any feature by any client. (Qualified calls only: $a.f$ (...))
The correctness of a class

- For every creation procedure \( cp \):
  \[
  \{ \text{pre}_{cp} \} \ \text{do}_{cp} \ \{ \text{post}_{cp} \ \text{and INV} \}
  \]

- For every exported routine \( r \):
  \[
  \{ \text{INV and pre}_r \} \ \text{do}_r \ \{ \text{post}_r \ \text{and INV} \}
  \]

- 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

\[(A1)\]

\[
\begin{align*}
\text{deposits} & \quad \text{deposits} \\
\text{withdrawals} & \quad \text{withdrawals}
\end{align*}
\]

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

\[(A2)\]

\[
\begin{align*}
\text{deposits} & \quad \text{deposits} \\
\text{withdrawals} & \quad \text{withdrawals}
\end{align*}
\]

\[
\text{balance}
\]
class ACCOUNT create

make

feature {NONE} -- Implementation

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

deposits: DEPOSIT_LIST
withdrawals: WITHDRAWAL_LIST
feature \{NONE\} -- Initialization

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

ensure
  balance_set: balance = initial_amount

end

feature -- Access

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

\texttt{deposit (sum: INTEGER) is}
\hspace{1em}-- Deposit \textit{sum} into the account.
\begin{itemize}
  \item\textbf{require}\hspace{1em}\texttt{not\_too\_small: sum \textgreater= 0}
\end{itemize}
\begin{itemize}
  \item\textbf{do}\hspace{1em}\texttt{add (sum)}
  \hspace{1em}\texttt{deposits.extend (create \{DEPOSIT\}.make (sum))}
\end{itemize}
\begin{itemize}
  \item\textbf{ensure}\hspace{1em}\texttt{increased: balance = old balance + sum}
\end{itemize}
\begin{itemize}
  \item\textbf{end}
\end{itemize}
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))
    end

    ensure
    decreased: balance = old balance – sum
    one_more: withdrawals.count = old withdrawals.count + 1
end
may_withdraw \( (\text{sum}: \text{INTEGER}) : \text{BOOLEAN} \) is
-- Is it permitted to withdraw \text{sum} from the
-- account?

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

\begin{itemize}
  \item \textbf{invariant}
  \begin{itemize}
    \item \textbf{not\_under\_minimum}: \text{balance} \geq \text{Minimum\_balance}
    \item \textbf{consistent}: \text{balance} = \text{deposits\_total} - \text{withdrawals\_total}
  \end{itemize}
\end{itemize}
The correctness of a class

- For every creation procedure $cp$:
  \[
  \{ \text{pre}_{cp} \} \text{ do}_{cp} \{ \text{post}_{cp} \text{ and INV} \}
  \]

- For every exported routine $r$:
  \[
  \{ \text{INV and pre}_r \} \text{ do}_r \{ \text{post}_r \text{ and INV} \}
  \]
feature {NONE} -- Initialization

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

Initial version
Correct version

```plaintext
feature \{NONE\} -- Initialization

make (initial_amount: INTEGER) is
    -- Set up account with initial_amount.
    require large_enough: initial_amount \geq\ Maximum_balance
    do
        create deposits.make
        create withdrawals.make
        deposit (initial_amount)
    end

ensure balance_set: balance = initial_amount
```
End of lecture 9