Programming in the large

Bertrand Meyer

Lecture 6: Design by Contract™

by Karine Arnout

Design by Contract

- A discipline of analysis, design, implementation, management

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>

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).

### Contracts for analysis

#### Deferred class

**PLAN**

**feature**

- `start_take_off` is
  - Initiate take-off procedures.
  - `deferred` ensures
  - `assigned_runway.is_clear`
  - `assigned_runway.owner = Current`

**end**

 chávez

**start_land**, `increase_altitude`, `decrease_altitude`, `moving`, `time_since_take_off`

**invariant**

- `assigned_runway.owner = Current`

**end**

#### Deferred class

**TANK**

**feature**

- `in_valve`, `out_valve`: VALVE

**require**

- `in_valve.open` and `out_valve.is_closed`

**deferred**

- `in_valve.is_closed` and `out_valve.is_closed`

**ensure**

- `is_full`

**invariant**

- `is_full` = `(gauge >= 0.97 * maximum) and (gauge <= maximum)`

### Contracts for analysis

#### Obligations

**Client**

- (Satisfy precondition:)
  - Make sure input valve is open, output valve is closed.

**Supplier**

- (Satisfy postcondition:)
  - Fill the tank and close both valves.

#### Benefits

**From precondition:**

- (Simpler processing thanks to assumption that valves are in the proper initial position.)

**From postcondition:**

- Get filled-up tank, with both valves closed.

### 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 in software

- Correctness is a relative notion: consistency of implementation vis-à-vis specification. (This assumes there is a specification!)
- Basic notation: \( \{P\} A \{Q\} \)
- "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 \( y \) ...

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

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

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>

### A class without contracts

**class** ACCOUNT

**feature** -- Access

\[ \text{balance: INTEGER} \]

-- Balance

\[ \text{Minimum\_balance: INTEGER is 1000} \]

-- Minimum balance

**feature** (NONE) -- Implementation of deposit and withdrawal

add (sum: INTEGER) is

-- Add sum to the balance (secret procedure).

do

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

end

### Introducing contracts

**class** ACCOUNT
create
make

**feature** (NONE) -- Initialization
make (initial\_amount: INTEGER) is

-- Set up account with initial\_amount.

require

\[ \text{large\_enough: initial\_amount} \geq \text{Minimum\_balance} \]

do

\[ \text{balance := initial\_amount} \]

ensure

\[ \text{balance\_set: balance} = \text{initial\_amount} \]

end

**feature** -- Deposit and withdrawal operations

deposit (sum: INTEGER) is

-- Deposit sum into the account.

do

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

end

withdraw (sum: INTEGER) is

-- Withdraw sum from the account.

do

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

end

may\_withdraw (sum: INTEGER): BOOLEAN is

-- Is it permitted to withdraw sum from the account?

do

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

end

**feature** -- Access

\[ \text{balance: INTEGER} \]

-- Balance

\[ \text{Minimum\_balance: INTEGER is 1000} \]

-- Minimum balance

**feature** (NONE) -- Implementation of deposit and withdrawal

add (sum: INTEGER) is

-- Add sum to the balance (secret procedure).

do

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

ensure

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

end

**feature** -- Deposit and withdrawal operations

deposit (sum: INTEGER) is

-- Deposit sum into the account.

require

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

do

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

ensure

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

end
Introducing contracts

withdraw \( (\text{sum}: \text{INTEGER}) \) is
-- Withdraw sum from the account.
require
not_too_small: \text{sum} \geq 0
not_too_big: \text{sum} \leq \text{balance} - \text{Minimum_balance}
do
add (- sum)
-- i.e. balance := balance - sum
ensure
deeased: \text{balance} = \text{old balance} - \text{sum}
end

The contract

\[
\begin{array}{|c|c|}
\hline
\text{withdraw} & \text{OBLIGATIONS} & \text{BENEFITS} \\
\hline
\text{Client} & \text{(Satisfy precondition:)} & \text{(From postcondition:)} \\
& \text{Make sure } \text{sum} \text{ is neither too small nor too big.} & \text{Get account updated with } \text{sum} \text{ withdrawn.} \\
\hline
\text{Supplier} & \text{(Satisfy postcondition:)} & \text{(From precondition:)} \\
& \text{Update account for withdrawal of } \text{sum}. & \text{Simpler processing: may assume } \text{sum} \text{ is within allowable bounds.} \\
\hline
\end{array}
\]

The imperative and the applicative

\[
\begin{array}{|c|c|}
\hline
\text{do} & \text{ensure} \\
\text{balance} := \text{balance} - \text{sum} & \text{balance} = \text{old balance} - \text{sum} \\
\hline
\text{PRESCRIPTIVE} & \text{DESCRIPTIVE} \\
\text{How?} & \text{What?} \\
\text{Operational} & \text{Denotational} \\
\text{Implementation} & \text{Query} \\
\text{Command} & \text{Expression} \\
\text{Instruction} & \text{Applicative} \\
\text{Imperative} & \\
\hline
\end{array}
\]

Introducing contracts

\[
\begin{array}{|c|c|}
\hline
\text{may_withdraw} \ (\text{sum}: \text{INTEGER}): \text{BOOLEAN} \ is
-- Is it permitted to withdraw \text{sum} from the account?
do
\text{Result} := (\text{balance} - \text{sum} \geq \text{Minimum_balance})
\hline
\text{invariant}
\text{not_under_minimum}: \text{balance} \geq \text{Minimum_balance}
\hline
\end{array}
\]

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: \text{x.f (...)}

The correctness of a class

- For every creation procedure \text{cp}:
  - \{Pre.\} \text{do}; \{Post. and INV\}
- For every exported routine \text{r}:
  - \{INV and Pre.\} \text{do}; \{Post. 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

\[(A1)\]
- \text{deposits}
- \text{withdrawals}

\[(A2)\]
- \text{deposits}
- \text{withdrawals}

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

A more sophisticated version

\begin{verbatim}
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.
  do balance := initial_amount
  create deposits.make
  create withdrawals.make
  ensure balance_set: balance = initial_amount
end

balance: INTEGER
-- Balance
Minimum_balance: INTEGER is 1000
-- Minimum balance

balance

feature -- Deposit and withdrawal operations
deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  require not_too_small: sum >= 0
  do add (sum)
  ensure increased: balance = old balance + sum
  one_more: deposits.count = old deposits.count + 1
end

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{verbatim}
**The correctness of a class**

- For every creation procedure \( cp \):
  \[
  \{ \text{Pre}_cp \} \text{ do } \{ \text{Post}_cp \text{ and } \text{INV} \}
  \]

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

---

**Initial version**

```plaintext
feature \{NONE\} -- Initialization
make \{initial_amount: INTEGER\} is
  require large_enough: \initial_amount \geq \text{Minimum_balance}
do
  balance := \initial_amount
create deposits.make
create withdrawals.make
ensure balance_set: balance = \initial_amount
end
```

**Correct version**

```plaintext
feature \{NONE\} -- Initialization
make \{initial_amount: INTEGER\} is
  require large_enough: \initial_amount \geq \text{Minimum_balance}
do
  create deposits.make
create withdrawals.make
deposit \{initial_amount\}
ensure balance_set: balance = \initial_amount
end
```