Object-Oriented Software Construction

Bertrand Meyer

Lecture 11:
Design by Contract™

Ariane-5 (Continued)

It was a REUSE error:
- The analysis was correct – for Ariane 4!
- The assumption was documented – in a design document!

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!

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.

**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

**Applications**

- Getting the software right
- Analysis
- Design
- Implementation
- Debugging
- Testing
- Management
- Maintenance
- Documentation

**The French Driver’s License issue**
**A human contract**

<table>
<thead>
<tr>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfy precondition:</td>
<td>Get package delivered by 10 a.m. next day.</td>
</tr>
<tr>
<td>Bring package before 4 p.m.; pay fee.</td>
<td></td>
</tr>
</tbody>
</table>

**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 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.

**Contracts for analysis**

```plaintext
def deferred class PLANE inherit AIRCRAFT
  feature
    start_take_off is
      require controls, passed assigned_runway, clear
      deferred
      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)
  ```
Contracts for analysis (cont’d)

defered class VAT inherit
  TANK
  feature
    ln_valve, out_valve: VALVE
  fill is
    -- Fill the vat.
    require
      ln_valve.open
      out_valve.closed
    deferred
      ensure
        ln_valve.closed
        out_valve.closed
        is_full
      end
    end
    empty, is_full, is_empty, gauge, maximum, ... [Other features]...
  invariant
    is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)
end

Precondition

-- i.e. specified only.
-- not specified.
Postcondition

Class invariant

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 (cont’d)

<table>
<thead>
<tr>
<th>fill</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 vat, with both valves closed.</td>
</tr>
<tr>
<td>Supplier</td>
<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>

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.

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?

Correctness in software

- 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 \(Q\).

Specifying a square root routine

\{x >= 0\}

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

\{abs (y ^ 2 - x) <= 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. \( \{\text{False}\} A \{\ldots\} \)
  2. \( \{\ldots\} A \{\text{True}\} \)

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 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

A class without contracts

class ACCOUNT feature -- Access
  balance: INTEGER
    -- Balance
  Minimum_balance: INTEGER is 1000
    -- Minimum balance
feature \( \{\text{NONE}\} \) -- 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 (initial_amount: INTEGER) is
  -- Set up account with initial_amount.
  do
    require
      large_enough: initial_amount >= Minimum_balance
    do
      balance := initial_amount
    ensure
      balance_set: balance = initial_amount
  end

---

Introducing contracts (cont’d)

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 (cont’d)

feature -- Deposit and withdrawal operations

deposit (sum: INTEGER) is
  -- Deposit sum into the account.
  do
    require
      not_too_small: sum >= 0
    do
      add (sum)
    ensure
      increased: balance = old balance + sum
  end

---
With contracts (cont’d)

```plaintext
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)
    -- i.e. balance := balance - sum
  ensure
    decreased: balance = old balance - sum
end
```

The imperative and the applicative

```plaintext
do
  balance := balance - sum
ensure
  balance = old balance - sum

<table>
<thead>
<tr>
<th>PRESRIPTIVE</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>
```

The contract

```
withdraw

<table>
<thead>
<tr>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td></td>
</tr>
<tr>
<td>(Satisfy precondition:) Make sure sum is neither too small nor too big.</td>
<td></td>
</tr>
<tr>
<td>(From postcondition:) Get account updated with sum withdrawn.</td>
<td></td>
</tr>
<tr>
<td>Supplier</td>
<td></td>
</tr>
<tr>
<td>(Satisfy postcondition:) Update account for withdrawal of sum.</td>
<td></td>
</tr>
<tr>
<td>(From precondition:) Simpler processing: may assume sum is within allowable bounds.</td>
<td></td>
</tr>
</tbody>
</table>
```

With contracts (end)

```plaintext
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: \( a.f (...) \))

Uniform Access

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

The correctness of a class

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

A more sophisticated version

```
class ACCOUNT create
  make
  feature {NONE} -- Implementation
    add (sum: INTEGER is
      add sum to the balance (secret procedure).
      balance := balance + sum
    ensure
      balance_increased: balance = old balance + sum
    end
    deposits: DEPOSIT_LIST
    withdrawals: WITHDRAWAL_LIST
  end
end
```
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
    end
  ensure balance_set: balance = initial_amount
end

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

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

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
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
The correctness of a class

- For every creation procedure `cp`:
  \{pre\} do \{post\} and INV

- For every exported routine `r`:
  \{INV and pre\} do \{post and INV\}

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

End of lecture 11