Object-Oriented Software Construction

Bertrand Meyer

Lecture 12:
Design by Contract™

The contract language

- Language of boolean expressions (plus old):
  - No predicate calculus (i.e. no quantifiers, ∀ or ∃).
  - Function calls permitted (e.g. in a STACK class):

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

remove (x: G) is
  -- Pop top of stack.
require
  not is_empty
do
...ensure
  not is_full
end
```

Contracts: run-time effect

- Compilation options (per class, in Eiffel):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions

The contract language (cont’d)

- First order predicate calculus may be desirable, but not sufficient anyway.
- Example: “The graph has no cycles”.
- In assertions, use only side-effect-free functions.
- Use of iterators provides the equivalent of first-order predicate calculus in connection with a library such as EiffelBase or STL. For example (Eiffel agents, i.e. routine objects):

```plaintext
my_integer_list.for_all (agent is_positive (?) )
```

with

```plaintext
is_positive (x: INTEGER): BOOLEAN is
do
  Result := (x > 0)
end
```
The imperative and the applicative

**do**

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

**ensure**

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

<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>

A contract violation is not a special case

- For special cases (e.g. “if the sum is negative, report an error…”)
  use standard control structures (e.g. if ... then ... else...).
- A run-time assertion violation is something else:
  the manifestation of

  **A DEFECT (“BUG”)**

What are contracts good for?

- Writing correct software (analysis, design, implementation, maintenance, reengineering).
- Documentation (the “contract” form of a class).
- Effective reuse.
- Controlling inheritance.
- Preserving the work of the best developers.
- Quality assurance, testing, debugging (especially in connection with the use of libraries).
- Exception handling.

Contracts and quality assurance

- Precondition violation: **Bug in the client**.
- Postcondition violation: **Bug in the supplier**.
- Invariant violation: **Bug in the supplier**.

\[\{ P \} A \{ Q \}\]
### Contracts and bug types

- Preconditions are particularly useful to find bugs in client code:
  
  ```
  class LIST[G]
  ...
  insert (x; G; i: INTEGER) is
  ensure
    i >= 0
    i <= count + 1
  end insert;
  ...
  your_list.insert (y, a + b + 1)
  ```

### Contracts missed

- Ariane 5 (see Jézéquel & Meyer, IEEE Computer, January 1997)
- Lunar Orbiter Vehicle
- Failure of air US traffic control system, November 2000
- Y2K
- etc. etc. etc.

### Contracts and quality assurance

- Use run-time assertion monitoring for quality assurance, testing, debugging.
- Compilation options (reminder):
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions

### Contracts and quality assurance

- Contracts enable QA activities to be based on a precise description of what they expect.
- Profoundly transform the activities of testing, debugging and maintenance.

> "I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. ... This sort of contract mechanism is the sine-qua-non of sensible software reuse."

Tom de Marco, IEEE Computer, 1997
Debugging with contracts: an example

- This example will use a live demo from EiffelStudio, with a "planted" error leading to a precondition violation.
- The example uses both the browsing and debugging mechanisms.

Linked list representation

Adding and catching a bug

- In class STARTER, procedure make_a_list, replace the first call to extend by a call to put.
- Execute system. What happens?
- Use browsing mechanisms to find out what's wrong (violated precondition).
- To understand, consider what the diagram of page 16 becomes when the number of list items goes to zero.
**Contract monitoring**

- Enabled or disabled by compile-time options.
- Default: preconditions only.
- In development: use “all assertions” whenever possible.
- During operation: normally, should disable monitoring. But have an assertion-monitoring version ready for shipping.
- Result of an assertion violation: exception.

- Ideally: static checking (proofs) rather than dynamic monitoring.

**Class example (cont’d)**

```plaintext
feature {NONE} -- Initialization
  make (initial_amount: INTEGER) is
    -- Set up account with initial_amount.
    require
      do
        large_enough: initial_amount >= Minimum_balance
        deposit (initial_amount)
        create deposits.make
        create withdrawals.make
    ensure
      balance_set: balance = initial_amount
  end

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

**Contracts and documentation**

Recall example class:

```plaintext
class ACCOUNT create
  make

feature {NONE} -- Implementation
  add (sum: INTEGER) is
    -- Add sum to the balance (secret procedure).
    do
      balance := balance + sum
    ensure
      increased: balance = old balance + sum
  end

deposits: DEPOSIT_LIST
withdrawals: WITHDRAWAL_LIST
```

**Class example (cont’d)**

```plaintext
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
```
Class example (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)
    withdrawals.extend (create (WITHDRAWAL).make (sum))
  ensure
    decreased: balance = old balance - sum
    one_more: withdrawals.count = old withdrawals.count + 1
end
```

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. (This raises a problem; can you see it?)

Class example (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
  consistent: balance = deposits.total - withdrawals.total
end
```

Export rule for preconditions

- In
  ```plaintext
  feature {A, B, C}
  r(...) is require
    some_property
  ```

- `some_property` must be exported (at least) to A, B and C!

- No such requirement for postconditions and invariants.
Contract form of ACCOUNT class

```plaintext
class interface ACCOUNT create
    make
feature
    balance: INTEGER
        -- Balance
    Minimum_balance: INTEGER is 1000
        -- Minimum balance
    deposit (sum: INTEGER)
        -- Deposit sum into the account.
        require
            not_too_small: sum >= 0
        ensure
            increased: balance = old balance + sum
end
```

Flat, interface

- **Flat form of a class**: reconstructed class with all the features at the same level (immediate and inherited). Takes renaming, redefinition etc. into account.
- The flat form is an inheritance-free client-equivalent form of the class.
- **Interface form**: the contract form of the flat form. Full interface documentation.

Contract form (cont’d)

```plaintext
withdraw (sum: INTEGER)
    require
        not_too_small: sum >= 0
        not_too_big: sum <= balance - Minimum_balance
    ensure
        decreased: balance = old balance - sum
        one_more: withdrawals.count = old withdrawals.count + 1
    may_withdraw (sum: INTEGER): BOOLEAN
        -- Is it permitted to withdraw sum from the
        -- account?

invariant
    not_under_minimum: balance >= Minimum_balance
    consistent: balance = deposits.total - withdrawals.total
```

Uses of the contract and interface forms

- Documentation, manuals
- Design
- Communication between developers
- Communication between developers and managers
Contracts and reuse

- The contract form — i.e. the set of contracts governing a class — should be the standard form of library documentation.

- Reuse without a contract is sheer folly.

- See the Ariane 5 example.

End of lecture 12