Six easy pieces

Bertrand Meyer

LASER, Biodola, September 2006
- Contracts and tests
- The cluster model
- Dealing with events
- Dealing with Void
- Open-sourcing EiffelStudio
- Towards an O-O process
Goals and techniques

Correctness
Robustness
Security
Extendibility
Reusability (consumer, producer)
“Maintainability”
Portability
Scalability

Classes
Objects
Deferred classes
Information hiding
Inheritance
Generics
Once routines
Agents
Renaming
GC
Multiple inheritance
Polymorphism
Constrained genericity
Dynamic binding
Exceptions
Deferred classes
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
Component = Contract + Implementation + Proof obligation
Software construction consists of building 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.
The three questions

- What does it expect?
- What does it promise?
- What does it maintain?
What we do with contracts today

Write better software (since we know what we are doing)
- Do serious analysis
- Do serious design
- Do serious reuse
- Do serious implementation
- Avoid bugs

Document software automatically
Help project managers do their job

(with run-time monitoring)

Perform systematic testing
Guide the debugging process
A class with contracts

class

    ACCOUNT

create

    make

feature {NONE} -- Initialization

    make (initial_amount: INTEGER)
    -- Set up account with initial_amount.

    require
        large_enough: initial_amount >= Minimum_balance
    do
        balance := initial_amount
    ensure
        balance_set: balance = initial_amount
end
A class with contracts

feature -- Access

balance: INTEGER
  -- Balance

Minimum_balance: INTEGER = 1000
  -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

add (sum: INTEGER)
  -- Add sum to the balance (secret procedure).
  do
    balance := balance + sum
  ensure
    increased: balance = old balance + sum
  end
A class with contracts

feature -- Deposit and withdrawal operations

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

-- Deposit \( \text{sum} \) into the account.

\[
\begin{align*}
\text{require} & \quad \text{not too small: sum } \geq 0 \\
\text{do} & \\
\text{add (sum)} \\
\text{ensure} & \quad \text{increased: balance } = \text{ old balance } + \text{ sum} \\
\text{end}
\end{align*}
\]
A class with contracts

\[\text{withdraw}(\text{sum} \colon \text{INTEGER})\]
\[\quad \text{-- Withdraw sum from the account.}\]
\[\text{require}\]
\[\quad \text{not\_too\_small} : \text{sum} \geq 0\]
\[\quad \text{not\_too\_big} : \text{sum} \leq \text{balance} - \text{Minimum\_balance}\]
\[\text{do}\]
\[\quad \text{add}(-\text{sum})\]
\[\text{ensure}\]
\[\quad \text{decreased} : \text{balance} = \text{old balance} - \text{sum}\]
\[\text{end}\]
A class with contracts (end)

```plaintext
may_withdraw \( \text{(sum: INTEGER): BOOLEAN} \)
   -- Is it permitted to withdraw \text{sum} from the
   -- account?
          do
              \text{Result} := (balance - sum} \geq \text{Minimum_balance})
         end

\textbf{invariant}

\textbf{not_under_minimum: balance} \geq \text{Minimum_balance}

end
```
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 \( r \):

\[ \{ \text{INV and pre}_r \} \text{ do } r \{ \text{post}_r \text{ and INV} \} \]

\[ \text{create } a.make (...) \]
deferred class \textit{VAT} inherit \textit{TANK}

feature

\textit{in\_valve}, \textit{out\_valve}: \textit{VALVE}

\textit{fill} is

\begin{verbatim}
  require
  \textit{in\_valve}.open
  \textit{out\_valve}.closed

  deferred
  ensure
  \textit{in\_valve}.closed
  \textit{out\_valve}.closed
  \textit{is\_full}

end
\end{verbatim}

\textit{empty, is\_full, is\_empty, gauge, maximum, ... [Other features] ...}

\textbf{Invariant}

\[ is\_full = (gauge \geq 0.97 \times \text{maximum}) \quad \text{and} \quad (gauge \leq 1.03 \times \text{maximum}) \]
Contracts and inheritance

Correct call in $C$:

\[
\text{if } a1.\alpha \text{ then } \\
a1.r(...)
\]

-- Here $a1.\beta$ holds

\[
\text{end}
\]
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
Contracts for testing and debugging

Contracts provide the right basis:
- Testing is there to find bugs
- A bug is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a bug:
- Precondition violation: bug in client
- Postcondition violation: bug in routine

In EiffelStudio: select compilation option for run-time contract monitoring at level of class, cluster or system.
Contract monitoring

“Development-driven test”
A revolutionary form of quality assurance
Moving the cursor forward

Before

Cursor

(index)

forth

"Biodola"

1

Count

After
Command “forth”

```
feature -- Status report

after: BOOLEAN
    -- Is there no valid cursor position to the right of cursor?

before: BOOLEAN
    -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

forth
    -- Move to next position; if no next position,
    -- ensure that `exhausted' will be true.

require -- from LINEAR
    not_after: not after
ensure then
    moved_forth: index = old index + 1
```
Where the cursor may go

Valid cursor positions
From the invariant of class LIST

```
invariant

prunable: prunable
before_definition: before = (index = 0)
after_definition: after = (index = count + 1)
    -- from CHAIN

non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

Valid cursor positions
Anecdotal & non-anecdotal evidence

HP 1:

\[
\text{invariant} \\
\quad r = 2^i
\]

HP 2: Eiffel messes up our great system!

Axa Rosenberg: postcondition fails in deep_clone of \textit{TWO\_WAY\_LIST}!

Patrice Chalin study (Concordia): Eiffel programmers do use contracts day in, day out.
AutoTest (Ilinca Ciupa, Andreas Leitner)

Integrated workbench for testing of components

Input: a set of classes

Push-button testing

Open-source; available for download from http://se.ethz.ch (source and binary)
Test automation: what does this mean?

- Test execution
- Robust execution
- Regression testing
- Test case generation (*test suites*)
- Test result verification (*test oracles*)
- Test scheduling
- Test case minimization
AutoTest architecture
Basic AutoTest scheme

- Test a set of classes completely automatically
- Generates instances and call their features with automatically selected arguments
- No manual test oracles: rely on contracts
  - Precondition violations: skip
  - Postcondition/invariant violation: bingo!
- Manual tests:
  - Can be added explicitly
  - Any test (manual or automatic) that fails becomes part of the test suite
Generation strategies

- For each type, create target pool:
  - Create instances using creation procedures (constructors)
  - This requires proper ordering of classes
- For each routine: create argument pool
- Invoke selected routines
AutoTest: strategies

- Argument and target generation: Adaptive Random Testing (see next)
- Precondition satisfaction: Planning, model checking
- Inheritance (“fragile base class” issue)
- Slicing
- If bug found: test minimization
Automatic Random Testing (Chen et al.)

Existing techniques: integers etc.

We generalized this to objects:
“object distance”
Object distance (Ilinca Ciupa)

\[ p \leftrightarrow q \Rightarrow \Delta \]

combination ( 

\text{type\_distance} (p\text{.}type, q\text{.}type), 
\text{field\_distance} (p, q), 
\text{recursive\_distance} ( 

\left\{ \left[ p\text{.}r \leftrightarrow q\text{.}r \right] \mid r \in \text{Reference\_attributes} \right\} )

)
When AutoTest finds bugs...

- All test execution is logged
- Results generated only after testing
- Bug found:
  - Minimize witness
  - Create manual test case
AutoTest demo
## Experimental results (random testing)

<table>
<thead>
<tr>
<th>Library</th>
<th>Failed tests/total</th>
<th>Buggy routines/ total</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>1513/39615 (3%)</td>
<td>127 / 1984 (6%)</td>
</tr>
<tr>
<td>Structures</td>
<td>1143 / 21242 (5%)</td>
<td>88 / 1400 (6%)</td>
</tr>
<tr>
<td>Gobo math</td>
<td>16/1539 (1%)</td>
<td>9 / 144 (6%)</td>
</tr>
<tr>
<td>DoctorC</td>
<td>1283 / 8972 (14%)</td>
<td>15 / 33 (45%)</td>
</tr>
</tbody>
</table>
AutoTest result analysis (Raluca Borga, U. of Cluj-Napoca, diploma work at ETH)

Fault classification (manual):

- **Specification**: discrepancy between contract and intention (as expressed e.g. by header comment)
- **Implementation**: contract looks OK, but routine body fails to meet it
- **Inheritance**
- **Don’t know**

Of those understood: about 50% implementation, 50% specification

- Results polluted by “void” issue (Will no longer exist thanks to attached types, see ECOOP 2005 paper)
Hermitage elevator
AutoTest developments

- Scaling up (memory, time)
- Complete integration with EiffelStudio environment
- Background, unobtrusive, continuous testing
- Automatic regression testing
- Distributed cooperative testing ("Testi@home")
Tests or proofs?

**Tests and proofs!**

**Examples:**

- Proving a class: attempt succeeds for some routines, fails for others
- Failed proof: proof obligations remain - candidates for tests
- Model checking counter-example: try to generate test data that exercises it
Tests and proofs

TAP
Tests And Proofs
ETH Zurich, 11-13 Feb 2007
CFP deadline: 30 September 06
http://tap.ethz.ch
Beefing up expressive power

Components to prove (e.g. EiffelBase)

Eiffel Model Library
Mathematical Model Library (MML)

Classes correspond to mathematical concepts:

\[ \text{SET}[G], \text{FUNCTION}[G, H], \text{TOTAL\_FUNCTION}[G, H], \text{RELATION}[G, H], \text{SEQUENCE}[G], \ldots \]

Completely applicative: no attributes (fields), no implemented routines (all completely deferred)

Specified with contracts (unproven) reflecting mathematical properties

Expressed entirely in Eiffel
Example MML class

class \textit{SEQUENCE} \langle G \rangle \textit{ feature}

\begin{verbatim}
count : NATURAL
   -- Number of items
last : G
   -- Last item

extended (x) : \textit{SEQUENCE} \langle G \rangle
   -- Identical sequence except x added at end
   ensure
     Result.count = count + 1
     Result.last = x
     Result.sub (1, count) ~ Current

mirrored : \textit{SEQUENCE} \langle G \rangle
   -- Same items in reverse order
   ensure
     Result.count = count
     ...
\end{verbatim}

Object equality
class LINKED_LIST[G]

feature

... remove_front

-- Remove first item

require

not empty

do

first := first . right

ensure

count = old count - 1

model = old model . tail

first = old item (2)

end ... end
Principles

Very simple mathematics only
- Logic
- Set theory
Components

Component = Contract + Implementation + Proof obligation
Design or verify?

The view that one can produce software using any technique and then “verify” it is a dangerous myth.

Contracts are one way to combine methodology (a priori) and verification (a posteriori).

Don’t take my word for it; try Eiffel.