Design by Contract™
Design by Contract

A discipline of analysis, design, implementation, management
Applications

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

- **Origin:** work on “axiomatic semantics” (Floyd, Hoare, Dijkstra), early seventies
- Some research languages had a built-in assertion mechanism: Euclid, Alphard
- Eiffel introduced the connection with object-oriented programming and made contracts a software construction methodology and an integral part of the language
- **Mechanisms for other languages:** Nana macro package for C++, JML for Java, Spec# (and dozens of others)
The basic idea

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
Documenting a program

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)?

Single product principle

The product is the software!
The contract 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
## A human contract

<table>
<thead>
<tr>
<th>deliver</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></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><strong>Supplier</strong></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>
## Contracts for analysis

<table>
<thead>
<tr>
<th>fill</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure input valve is open, output valve closed</td>
<td>(From postcondition:) Get filled-up tank, with both valves closed</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Fill the tank 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>
Correctness in software

Correctness is a relative notion: consistency of implementation with specification

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

\[
\{P\} \ A \ \{Q\}
\]

“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 * \text{epsilon} * y \}

-- i.e.: \( y \) approximates exact square root of \( x \)
-- within \( \text{epsilon} \)
Software correctness (a quiz)

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. \( \{\ldots\} \ A \ \{\text{True}\} \)
A contract (from EiffelBase)

\[
\text{extend} \ (\text{new} : G ; \text{key} : H) \\
\quad -- \text{Assuming there is no item of key key,} \\
\quad -- \text{insert new with key; set inserted.}
\]

require

key_not_present: \text{not has (key)}

ensure

insertion_done: \text{item (key) = new}
key_present: \text{has (key)}
inserted: \text{inserted}
one_more: \text{count = old count + 1}
## The contract

<table>
<thead>
<tr>
<th>Routine</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>PRECONDITION</td>
<td>POSTCONDITION</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>POSTCONDITION</td>
<td>PRECONDITION</td>
</tr>
</tbody>
</table>
A class without contracts

\[
\text{class} \quad \text{ACCOUNT} \\
\text{feature} \quad \text{-- Access} \\
\quad \text{balance: INTEGER} \\
\quad \quad \text{-- Balance} \\
\quad \text{Minimum_balance: INTEGER = 1000} \\
\quad \quad \text{-- Minimum balance} \\
\text{feature \{NONE\} \quad \text{-- Deposit and withdrawal} } \quad \text{add (sum: INTEGER)} \\
\quad \quad \text{-- Add sum to the balance.} \\
\quad \quad \quad \text{do} \\
\quad \quad \quad \quad \text{balance := balance + sum} \\
\quad \quad \text{end}
\]
A class without contracts

feature -- Deposit and withdrawal operations

    deposit (sum : INTEGER)
      -- Deposit sum into the account.
      do
        add (sum)
      end

    withdraw (sum : INTEGER)
      -- Withdraw sum from the account.
      do
        add (- sum)
      end

    may_withdraw (sum : INTEGER): BOOLEAN
      -- 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)
-- Set up account with initial_amount.

require
large_enough: initial_amount >= Minimum_balance

do

balance := initial_amount

ensure
balance_set: balance = initial_amount
end
Introducing contracts

**feature** -- Access

\[ \text{balance: INTEGER} \]
\[ \quad \text{-- Balance} \]

\[ \text{Minimum\_balance: INTEGER = 1000} \]
\[ \quad \text{-- Lowest permitted balance} \]

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

\[ \text{add (sum: INTEGER)} \]
\[ \quad \text{-- Add sum to the balance.} \]
\[ \quad \text{do} \]
\[ \quad \quad \text{balance := balance + sum} \]
\[ \quad \text{ensure} \]
\[ \quad \quad \text{increased: balance = old balance + sum} \]
\[ \quad \text{end} \]
Introducing contracts

feature -- Deposit and withdrawal operations

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

require
not_too_small: sum >= 0

postcondition
increased: balance = old balance + sum
Introducing contracts

\textit{withdraw}(\textit{sum}: \text{INTEGER})

\text{-- Withdraw \textit{sum} from the account.}

\textbf{require}
\begin{itemize}
  \item \textit{not\_too\_small}: \textit{sum} \geq 0
  \item \textit{not\_too\_big}: \textit{sum} \leq \textit{balance} - \text{Minimum\_balance}
\end{itemize}

\textbf{do}
\begin{itemize}
  \item \textit{add}(-\textit{sum})
\end{itemize}

\text{-- i.e. } \textit{balance} := \textit{balance} - \textit{sum}

\textbf{ensure}
\begin{itemize}
  \item \textit{decreased}: \textit{balance} = \text{old balance} - \textit{sum}
\end{itemize}

\textbf{end}

\text{Value of } \textit{balance}, \text{ captured on entry to routine}
### The contract

<table>
<thead>
<tr>
<th>withdraw</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>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>( balance := balance - \text{sum} )</td>
<td>( balance = \text{old balance} - \text{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>
Introducing contracts

may_withdraw (sum: INTEGER): BOOLEAN
    -- Is it permitted to withdraw sum from 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: \( x.f(...) \)
For every creation procedure \( cp \):

\[
\{\text{Pre}_{cp}\} \textit{ do } cp \{\text{INV and Post}_{cp}\}
\]

For every exported routine \( r \):

\[
\{\text{INV and Pre}_{r}\} \textit{ do } r \{\text{INV and Post}_{r}\}
\]
Uniform Access

\[ \text{balance} = \text{deposits}.\text{total} - \text{withdrawals}.\text{total} \]
A slightly more sophisticated version

class ACCOUNT
create
make
feature {NONE} - Implementation

add (sum : INTEGER)
-- Add sum to the balance.
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)
        -- 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
feature -- Access
    balance: INTEGER
        -- Balance
    Minimum_balance: INTEGER = 1000
        -- Minimum balance
New version

**feature** -- Deposit and withdrawal operations

```plaintext
deposit (sum: INTEGER)
  -- 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
    one_more: deposits.count = old deposits.count + 1
end
```

withdraw (sum : INTEGER)
  -- 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
may_withdraw (sum : INTEGER): BOOLEAN
    -- Is it permitted to withdraw sum from 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 \( cp \):

\[
\{ \text{Pre}_{cp} \} \text{ do}_{cp} \{ \text{INV and Post}_{cp} \}
\]

For every exported routine \( r \):

\[
\{ \text{INV and Pre}_r \} \text{ do}_r \{ \text{INV and Post}_r \}
\]
Uniform Access

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

feature {NONE} - Initialization

make (initial_amount: INTEGER)
  -- Set up account with initial_amount.
  require
  large_enough: initial_amount >= Minimum_balance
  do
    create deposits.make
    create withdrawals.make
    balance := initial_amount
    deposit (initial_amount)
  ensure
    balance_set: balance = initial_amount
end
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
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, such as if ... then ... else...

A run-time assertion violation is something else: the manifestation of

A DEFECT (“BUG”)
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: run-time effect

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

Contracts express implicit assumptions behind code
- A bug is a discrepancy between intent and code
- Contracts state the intent!

In EiffelStudio: select compilation option for run-time contract monitoring at level of:
- Class
- Cluster
- System

May disable monitoring when releasing software
A revolutionary form of quality assurance
Contract monitoring

A contract violation always signals a bug:

- **Precondition violation:** bug in **client**
- **Postcondition violation:** bug in **routine**