Design by Contract

- A discipline of analysis, design, implementation, management
Design by Contract

- 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.

A human contract

<table>
<thead>
<tr>
<th></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>

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

defferred class PLANE
  feature start_take_off
    require controls.passed assigned_runway.is_clear
    deferred ensure assigned_runway.owner = Current
    end
  start_landing, increase_altitude, decrease_altitude, moving,
  altitude, speed, time_since_take_off
  invariant
    time_since_take_off <= 20
    assigned_runway.owner = Current
    moving = (speed > 10)
end

Contracts for analysis

defered class TANK
  feature in_valve, out_valve: VALVE
    require in_valve.open out_valve.is_closed
    deferred ensure in_valve.is_closed
    out_valve.is_closed
    is_full
    empty, is_full, is_empty, gauge, maximum, ...
    invariant
      is_full = (gauge >= 0.97 * maximum) and (gauge <= maximum)
    end

Contracts for analysis

<table>
<thead>
<tr>
<th></th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td><em>(Satisfy precondition:)</em> Make sure input valve is open, output valve is closed.</td>
<td><em>(From postcondition:)</em> Get filled-up tank, with both valves closed.</td>
</tr>
<tr>
<td>Supplier</td>
<td><em>(Satisfy postcondition:)</em> Fill the tank and close both valves.</td>
<td><em>(From precondition:)</em> Simpler processing thanks to assumption that valves are in the proper initial position.</td>
</tr>
</tbody>
</table>

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: \( (P, Q) \): 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 \( Q \).

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 \cdot \text{epsilon} \cdot 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} \}

A contract (from EiffelBase)

```eiffel
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

```plaintext
class ACCOUNT
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
        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
end
```
Introducing contracts

class ACCOUNT
create
make
feature {NONE} -- Initialization
make (initial_amount: INTEGER) is
  require
  large_enough: initial_amount >= Minimum_balance
do
  balance := initial_amount
ensure
  balance_set: balance = initial_amount
end

feature -- Access
balance: INTEGER
  -- Balance
Minimum_balance: INTEGER is 1000
  -- Minimum balance
feature {NONE} -- Implementation of deposit and withdrawal
add (sum: INTEGER) is
  do
    balance := balance + sum
  ensure
    increased: balance = old balance + sum
end

feature -- Deposit and withdrawal operations
deposit (sum: INTEGER) is
  require
    not_too_small: sum >= 0
do
  add (sum)
ensure
  increased: balance = old balance + sum
end
Introducing contracts

```plaintext
withdraw (sum: INTEGER) is
  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 contract

```
withdraw

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

The imperative and the applicative

```
do
  balance := balance - sum
  ensure
    balance = old balance - 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>
```
Introducing contracts

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: x.f (...))

The correctness of a class

- For every creation procedure cp:
  {Prec} do. {Post and INV}
- For every exported routine r:
  {INV and Pre} 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**

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

**A more sophisticated version**

```plaintext
class ACCOUNT
create
make
feature (NONE) -- Implementation
add (sum: INTEGER) is
  do balance := balance + sum
  ensure balance_increased: balance = old balance + sum
end

create deposits: DEPOSIT_LIST
create withdrawals: WITHDRAWAL_LIST
```

**A more sophisticated version**

```plaintext
feature (NONE) -- Initialization
make (initial_amount: INTEGER) is
  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 is 1000
  -- Minimum balance
```
A more sophisticated version

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
    one_more: deposits.count = old deposits.count + 1
end

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

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 \( cp \):
  \[ \{\text{Pre}_\text{cp}\} \quad \text{do} \quad \{\text{Post}_\text{cp} \quad \text{and} \quad \text{INV}\} \]

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

---

Initial version

```
feature {NONE} -- Initialization
make (initial_amount: INTEGER) is
  require large_enough: initial_amount >= Minimum_balance
do
  balance := initial_amount
  create deposits.make
  create withdrawals.make

  ensure balance_set: balance = initial_amount
end
```

---

Correct version

```
feature {NONE} -- Initialization
make (initial_amount: INTEGER) is
  require large_enough: initial_amount >= Minimum_balance
do
  create deposits.make
  create withdrawals.make
  deposit (initial_amount)

  ensure balance_set: balance = initial_amount
end
```
End of lecture 6