Trusted Components

Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 8: Design by Contract (1/2)
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
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>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>
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).
No hidden clauses

- In

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

- `some_property` must be exported (at least) to classes `A, B` and `C`!
- No such requirement for postconditions and invariants.
deferred class

PLANE

feature

start_take_off is -- Initiate take-off procedures.
require
controls.passed
assigned_runway.is_clear

defered
ensure
assigned_runway.owner = Current
moving

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)

end
deferred class

TANK

feature

in_valve, out_valve: VALVE

fill is

require

in_valve.open
out_valve.is_closed

end

deferred

ensure

in_valve.is_closed
out_valve.is_closed
is_full

end

Precondition

-- Fill the tank.

-- i.e. specified only.

-- not implemented.

Postcondition

empty, is_full, is_empty, gauge, maximum, ... [Other features] ...

Precondition

Postcondition

Class invariant

invariant

is_full = (gauge >= 0.97 * maximum) and (gauge <= maximum)

end
### Contracts for analysis

<table>
<thead>
<tr>
<th>fill</th>
<th><strong>OBLIGATIONS</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure input valve is open, output valve is 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>
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.
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
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:
  - \textit{Strongest} postcondition (from given precondition).
  - \textit{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 \times \text{epsilon} \times y\}

-- i.e.: \( y \) approximates exact square root of \( x \)

-- within \text{epsilon}
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. \(\{False\} \ A \ \{...\}\)
2. \(\{...\} \ A \ \{True\}\)
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
```
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>
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
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
    do
      balance := balance + sum
    end
A class without contracts

**feature** -- Deposit and withdrawal operations

```plaintext
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
end
```
class ACCOUNT

create

make

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
  ensure
    balance_set: balance = initial_amount
end
Introducing contracts

**feature** -- Access

\[\begin{align*}
    &balance: \text{INTEGER} \\
    &\quad \text{-- Balance}
\end{align*}\]

\[\begin{align*}
    &\text{Minimum\_balance: INTEGER is 1000} \\
    &\quad \text{-- Minimum balance}
\end{align*}\]

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

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

feature -- Deposit and withdrawal operations

\[\text{deposit } (\text{sum}: \text{INTEGER}) \text{ is} \]
\[\quad \text{-- Deposit sum into the account.} \]
\[\quad \text{require} \]
\[\quad \quad \text{not}\_\text{too}\_\text{small}: \text{sum} \geq 0 \]
\[\quad \text{do} \]
\[\quad \quad \text{add } (\text{sum}) \]
\[\quad \text{ensure} \]
\[\quad \quad \text{increased}: \text{balance} = \text{old balance} + \text{sum} \]
\[\text{end} \]
Introducing contracts

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 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 - sum ]</td>
<td>[ balance = \textbf{old} balance - sum ]</td>
</tr>
<tr>
<td>PRESCRIPTIVE</td>
<td>DESCRIPTIVE</td>
</tr>
<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.
  - Before and after execution of any feature by any client. (Qualified calls only: \( x.f (...) \))
The correctness of a class

- For every creation procedure \( cp \):
  \[
  \{ \text{Pre}_{cp} \} \ do_{cp} \ \{ \text{Post}_{cp} \ \text{and} \ \text{INV} \}
  \]

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

- 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

(A1)

\[
\begin{align*}
\text{deposits} & \quad \text{deposits} \quad \text{deposits} \\
\text{withdrawals} & \quad \text{withdrawals} \quad \text{withdrawals}
\end{align*}
\]

(A2)

\[
\begin{align*}
\text{deposits} & \quad \text{deposits} \quad \text{deposits} \\
\text{withdrawals} & \quad \text{withdrawals} \quad \text{withdrawals}
\end{align*}
\]

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

ACCOUNT

create

make

feature \{NONE\} -- Implementation

\textit{add} (sum: INTEGER) \textbf{is}

\textit{do}

\quad balance := balance + sum

\textit{ensure}

\quad balance\_increased: balance = old balance + sum

\textit{end}

deposits: DEPOSIT\_LIST

withdrawals: WITHDRAWAL\_LIST
A more sophisticated 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
   end

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
A more sophisticated version

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
A more sophisticated version

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

- For every exported routine \( r \):
  \[
  \{ \text{INV} \ \text{and} \ \text{Pre}_r \} \ \text{do}_r \ \{ \text{Post}_r \ \text{and} \ \text{INV} \}
  \]
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
    ensure
        balance_set: balance = initial_amount
end

Violates the invariant: balance = deposits.total – withdrawals.total
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
  end

ensure
  balance_set: balance = initial_amount
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
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")
Remember the Hoare triples for class correctness:

- \{\text{Pre}_{cp}\} \text{ do}_{cp} \{\text{Post}_{cp} \text{ and INV}\}
- \{\text{INV and Pre}_{r}\} \text{ do}_{r} \{\text{Post}_{r} \text{ and INV}\}

- Precondition violation: Bug in the client.
- Postcondition violation: Bug in the supplier.
- Invariant violation: Bug in the supplier.
Contracts and bug types

- Preconditions are particularly useful to find bugs in client code:

```
your_list.insert (y, a + b + 1)
```

```
class LIST [G]
...
insert (x: G; i: INTEGER) is
  require
    i >= 0
    i <= count + 1
```
Contracts and quality assurance

- Use run-time assertion monitoring for:
  - quality assurance
  - testing
  - debugging

- Compilation options:
  - No assertion checking
  - Preconditions only
  - Preconditions and postconditions
  - Preconditions, postconditions, class invariants
  - All assertions
Run-time assertion monitoring

- Enabled / disabled through compiler options

- Methodology:
  - During development: “all assertions” whenever possible
  - During operation: disable assertion monitoring

- Result of an assertion violation:
  - Exception

- Ideally:
  - Static checking rather than dynamic monitoring
Example delivered with EiffelStudio:

$ISE_EIFFEL\examples\studio\objtour$

Class \textit{STARTER}, procedure \textit{make\_a\_list}:

\begin{verbatim}
make_a_list (first, second, third: REAL) is
  -- Create my_list with the three given values.
  do
    create my_list.make
    my_list.extend (first)
    my_list.extend (second)
    my_list.extend (third)
  end

with

my_list: LINKED\_LIST [REAL]
\end{verbatim}

What about replacing \textit{extend} by \textit{put}?
class LINKED_LIST

... feature -- Element change

    extend (v: like item)
        -- Add v to end. Do not move cursor.

    put (v: like item)
        -- Replace current item by v.
        -- (Synonym for replace)

... end

No current item: The list is empty!
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
“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
Ariane 5: $500 million (not insured) was a REUSE error:

- Exception caused by a 64-bit real value incorrectly translated into a 16-bit integer.
- Analysis had “proved” that the exception could not occur! Was correct ... for Ariane 4!
- The assumption was documented in a design document, not in the code...
- With assertions, the error would almost certainly have been detected.

“Reuse without contracts is sheer folly!”
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
Contracts and inheritance

- **Issues:** what happens, under inheritance, to
  - Class invariants?
  - Routine preconditions and postconditions?
Invariant inheritance rule:

The invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed.
Contracts and inheritance

Correct call:

```plaintext
if a1.α then
  a1.r (...)
  -- Here a1.β holds.
end
```

`a1: A`

...  

`a1.r (...)`

`C`  

`A`

`D`

`B`

`r is require α ensure β`

`r is require γ ensure δ`
When redeclaring a routine:
- Precondition may only be kept or weakened.
- Postcondition may only be kept or strengthened.

Redeclaration covers both redefinition and effecting.
Assertion redeclaration rule in Eiffel

- Redefined version may **not** have `require` or `ensure`.

- May have nothing (assertions kept by default), or `require` else `new_pre` `ensure` then `new_post`

- Resulting assertions are:
  ```plaintext
ew_pre or else original_precondition original_postcondition and then new_post
```
Agenda for today

- Design by Contract: definition and benefits
- Correctness in software
- Example: Adding contracts to an existing class
- Debugging with contracts
- Contracts and reuse
- Contracts and inheritance
- Methodology
Methodological notes

- Contracts are not input checking tests...
- ... but they can be used to help weed out undesirable input.
- Filter modules:
Precondition design

- Clients must guarantee precondition before the call.
- Not necessarily through testing:

  Scheme 1 (testing):

  ```
  if not my_stack.is_full then
    my_stack.put (some_element)
  end
  ```

  Scheme 2 (guaranteeing without testing):

  ```
  my_stack.remove
  ...
  my_stack.put (some_element)
  ```
Another example

\[ \text{sqrt} (x, \text{epsilon} : \text{REAL}) : \text{REAL} \text{ is} \]
--- Square root of \( x \), precision \( \text{epsilon} \)

\text{require}

\begin{align*}
\text{x\_non\_negative} & : x \geq 0 \\
\text{epsilon\_non\_negative} & : \text{epsilon} \geq 0
\end{align*}

\text{do}

\ldots

\text{ensure}

\text{sqrt\_within\_requested\_precision}:

\begin{align*}
\text{abs} (\text{Result} \wedge 2 - x) & \leq 2 \times \text{epsilon} \times \text{Result}
\end{align*}

\text{end}
## The contract

<table>
<thead>
<tr>
<th>sqrt</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Provide non-negative value and precision that is not too small.</td>
<td>(From postcondition:) Get square root within requested precision.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Produce square root within requested precision.</td>
<td>(From precondition:) Simpler processing thanks to assumptions on value and precision.</td>
</tr>
</tbody>
</table>
No defensive programming!

- It is never acceptable to have a routine of the form

```pascal
sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
    x >= 0
    epsilon >= 0
  do
    if x < 0 then
      ... Do something about it (?) ...
    else
      ... normal square root computation ...
    end
  ensure
    abs (Result ^ 2 - x) <= 2 * epsilon * Result
  end
```

Chair of Software Engineering
No defensive programming

- For every consistency condition that is required to perform a certain operation:
  - Assign responsibility for the condition to one of the contract’s two parties (supplier, client).
  - Stick to this decision: do not duplicate responsibility.

- Simplifies software and improves global reliability.
How strong should a precondition be?

- Two opposite styles:
  - Tolerant: weak preconditions (including the weakest, True).
  - Demanding: strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call.

- Partly a matter of taste.

- But: demanding style leads to a better distribution of roles, provided the precondition is:
  - Justifiable in terms of the specification only.
  - Documented (through the short form).
  - Reasonable!
A demanding style

\[ \text{sqrt} \ (x, \ \text{epsilon}: \ \text{REAL}): \ \text{REAL} \ \text{is} \]
\[ \quad \text{-- Square root of } x, \text{ precision epsilon} \]
\[ \quad \text{-- Same version as before} \]

\begin{align*}
\text{require} & \\
& x \geq 0 \\
& \text{epsilon} \geq 0
\end{align*}

\begin{align*}
\text{do} & \\
& \ldots \\
\text{ensure} & \\
& \text{abs} \ (\text{Result}^2 - x) \leq 2 \times \text{epsilon} \times \text{Result}
\end{align*}

\text{end}
A tolerant style

\[ \text{sqrt}(x, \epsilon: \text{REAL}): \text{REAL} \text{ is} \]

-- Square root of \( x \), precision \( \epsilon \)

require

\textbf{True}

do

\textbf{if} \ x < 0 \ \textbf{then}

... Do something about it (?) ...

\textbf{else}

... normal square root computation ...

\textit{computed} := \textbf{True}

ensure

\textbf{computed} \ \textbf{implies}

\[ \text{abs}(\text{Result}^2 - x) \leq 2 \times \epsilon \times \text{Result} \]

end
Contrasting styles

\[
\text{demanding\_put } (x: G) \text{ is} \\
\quad \text{-- Push } x \text{ on top of stack.} \\
\text{require} \\
\quad \text{not } is\_full \\
\text{do} \\
\quad \text{....} \\
\text{end}
\]

\[
\text{tolerant\_put } (x: G) \text{ is} \\
\quad \text{-- Push } x \text{ if possible, otherwise set impossible to } \text{true.} \\
\text{do} \\
\quad \text{if not } is\_full \text{ then} \\
\quad \quad put (x) \\
\quad \text{else} \\
\quad \quad \text{impossible } := \text{True} \\
\text{end}
\]
Invariants and business rules

- Invariants are absolute consistency conditions.
- They can serve to represent business rules if knowledge is to be built into the software.

- Form 1

  ```
  invariant
  not_under_minimum: balance >= Minimum_balance
  ```

- Form 2

  ```
  invariant
  not_under_minimum_if_normal:
  normal_state implies
  (balance >= Minimum_balance)
  ```
Complementary material

- **OOSC2:**
  - Chapter 11: Design by Contract: building reliable software
  - Chapter 16: Inheritance techniques
    - 16.1 Inheritance and assertions

- **About Ariane 5:**
End of lecture 8