Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 9: Design by Contract (2/2)
Agenda for today

- Contracts for documentation
- Other kinds of assertions
- Exception handling
**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.
  - Keep contracts (preconditions, postconditions, class invariant).
  - Remove any contract clause that refers to a secret feature.
class ACCOUNT
create make

feature {NONE} -- Implementation

add (sum: INTEGER) is
  -- Add sum to the balance (secret procedure).
  do
    balance := balance + sum
  ensure
    balance_increased: balance = old balance + sum
  end

deposits: DEPOSIT_LIST
withdrawals: WITHDRAWAL_LIST
Class ACCOUNT (2/5)

feature {NONE} -- Initialization

make (initial_amount: INTEGER) is
    -- Set up account with initial_amount.
    require
        large_enough: initial_amount >= Minimum_balance
    do
        create deposits.make
        create withdrawals.make
        deposit (initial_amount)
    ensure
        balance_set: balance = initial_amount
end

feature -- Access

balance: INTEGER
    -- Balance

Minimum_balance: INTEGER is 1000
    -- Minimum balance
Class **ACCOUNT** (3/5)

**feature** -- Deposit and withdrawal operations

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

\[
\begin{align*}
\text{-- Deposit sum into the account.} \\
\text{require} \\
\text{not\_too\_small: sum} & \geq 0 \\
\text{do} \\
\text{add (sum)} \\
\text{deposits.extend (create \{DEPOSIT\}.make (sum))} \\
\text{ensure} \\
\text{increased: balance = old balance + sum} \\
\text{one\_more: deposits.count = old deposits.count + 1} \\
\end{align*}
\]

end
Class ACCOUNT (4/5)

```haskell
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
```
Class `ACCOUNT` (5/5)

```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
```
class interface

ACCOUNT

create

make

feature -- Access

balance: INTEGER
    -- Balance

Minimum_balance: INTEGER is 1000
    -- Minimum balance

feature -- Deposit and withdrawal operations

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

    require not_too_small: sum >= 0

    ensure increased: balance = old balance + sum

end

No keyword “is”

No keyword “end”
Contract form of class **ACCOUNT (2/2)**

```plaintext
withdraw (sum: INTEGER)
   -- Withdraw sum from the account.
   require
      not_too_small: sum >= 0
      not_too_big: sum <= balance - Minimum_balance
   ensure
      decreased: balance = old balance - sum

may_withdraw (sum: INTEGER): BOOLEAN
   -- Is it permitted to withdraw sum from the account?

invariant
   not_under_minimum: balance >= Minimum_balance

end
```
Flat and interface views of a class

- Flat form of a class:
  - Reconstructed class with all features at the same level (immediate + inherited).
  - Takes renaming, redefinition, ... into account.

- Interface form:
  - Contract form of the flat form.
  - Full interface documentation.
Contract and interface forms: what for?

- Documentation
- Design
- Communication
  - between developers
  - between developers and managers
Agenda for today

- Contracts for documentation
- Other kinds of assertions
- Exception handling
Loop trouble

- Loops are needed, powerful
- But very hard to get right:
  - “off-by-one”
  - Infinite loops
  - Improper handling of borderline cases
The answer: assertions

- Use of loop variants and invariants.

- A loop is a way to compute a certain result by successive approximations (e.g. computing the maximum value of an array of integers)
Loop as approximation strategy

Result = s1

Result = max (s1, s2)

Result = max (s1, s2, …, si)

Result = max (s1, s2, …, si, …, sn)
Computing the max of an array

- Approach by successive slices:

```haskell
max_of_array (t: ARRAY [INTEGER]): INTEGER is
   -- Maximum value of array t
   require
t_not_void: t /= Void
t_not_empty: not t.is_empty
local
   i: INTEGER
do
   from
      i := t.lower
      Result := t @ t.lower
   until
      i = t.upper
   loop
      i := i + 1
      Result := Result.max (t @ i)
end
```

Loop variants and invariants

- Syntax:

```
from initialization
invariant
  loop invariant -- Correctness property
variant
  loop variant -- Ensure loop termination.
until
  exit condition
loop
  body
end
```
Loop invariant

(Do not confuse with class invariant)

- Boolean property that must be:
  - Satisfied after initialization (from clause)
  - Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied
Loop variant

- Integer expression that must:
  - Be non-negative after initialization (from clause)
  - Decrease (by at least one), while remaining non-negative, for every iteration of the body (loop clause) executed with exit condition not satisfied
Maximum of an array with loop assertions

\[
\text{max_of_array} \ (t: \ ARRAY \ [INTEGER]): \ INTEGER \ \text{is}
\]
\[
\quad \text{-- Maximum value of array } t
\]
\[
\text{require}
\]
\[
\quad \text{t_not_void: } t \neq \text{Void}
\]
\[
\quad \text{t_not_empty: } \text{not } t . \text{is_empty}
\]
\[
\text{local}
\]
\[
\quad i: \ INTEGER
\]
\[
\text{do}
\]
\[
\quad \text{from}
\]
\[
\quad \quad i := t . \text{lower}
\]
\[
\quad \quad \textbf{Result} := t @ t . \text{lower}
\]
\[
\quad \textbf{invariant}
\]
\[
\quad \quad \text{-- Result is the max of the elements of } t \text{ at indices}
\]
\[
\quad \quad \text{-- } t . \text{lower to } i
\]
\[
\quad \textbf{variant}
\]
\[
\quad \quad t . \text{upper} - i
\]
\[
\quad \textbf{until}
\]
\[
\quad \quad i = t . \text{upper}
\]
\[
\quad \textbf{loop}
\]
\[
\quad \quad i := i + 1
\]
\[
\quad \quad \textbf{Result} := \textbf{Result}.\text{max} \ (t @ i)
\]
\[
\quad \textbf{end}
\]
\[
\text{end}
\]
Power of the assertion language

- Assertion language:
  - Not first-order predicate calculus
  - But powerful through: Function calls
  - Even allows to express: Loop properties
Functions in contracts

- Example: In a *STACK* class:

- **put** \( (x: G) \) is
  -- Push \( x \) on top of stack.
  
  \[
  \text{require} \quad \text{not is}\_\text{full}
  \]
  
  \[
  \text{do}
  \]
  
  ... 
  
  \[
  \text{ensure} \quad \text{not is}\_\text{empty}
  \]
  
  \[
  \text{end}
  \]

- **remove** \( (x: G) \) is
  -- Pop top of stack.
  
  \[
  \text{require} \quad \text{not is}\_\text{empty}
  \]
  
  \[
  \text{do}
  \]
  
  ... 
  
  \[
  \text{ensure} \quad \text{not is}\_\text{full}
  \]
  
  \[
  \text{end}
  \]

In assertions, use only side-effect-free functions.
Agents in contracts

- Use of iterators (in class \textit{LINEAR}) provides the equivalent of first-order predicate calculus:

\[
\text{my\_integer\_list\_for\_all} \ (\text{agent} \ \text{is\_positive} \ (?) )
\]

with

\[
\text{is\_positive} \ (x: \text{INTEGER}): \text{BOOLEAN} \ \text{is}
\]

\[
\text{do}
\]

\[
\text{Result} := (x > 0)
\]

\[
\text{end}
\]
**Check instruction**

- To ensure that a **property is true at a certain point of the routine execution**.

- Similar to “assert” in C/C++.

- You may add **check** clauses for readability when using a tolerant programming style:

  ```
  my_stack.remove
  check
    my_stack_not_full: not my_stack.is_full
  end
  my_stack.put (some_element)
  ```
Agenda for today

- Contracts for documentation
- Other kinds of assertions
- Exception handling
The need for exceptions arises when the contract is broken.

Two concepts:

- **Failure**: a routine, or other operation, is unable to fulfill its contract.
- **Exception**: an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.
The original strategy

\[ r (\ldots) \text{ is} \]
\[
\text{require} \quad \ldots
\]
\[
\begin{align*}
\text{do} \\
& \quad op_1 \\
& \quad op_2 \\
& \quad \ldots \\
& \quad op_i \\
& \quad \ldots \\
& \quad op_n \\
\text{ensure} \quad \ldots
\end{align*}
\]
\[
\text{end}
\]

\text{Fails, triggering an exception in} \ r \ (r \text{ is recipient of exception}).
Causes of exceptions

- Assertion violation
- Void call ($x.f$ with no object attached to $x$)
- Operating system signal (arithmetic overflow, no more memory, interrupt ...)

Chair of Software Engineering

Trusted Components: Reuse, Contracts and Patterns - Lecture 9
Handling exceptions properly

- Safe exception handling principle:
  - There are only two acceptable ways to react for the recipient of an exception:
    - Concede failure, and trigger an exception in the caller (**Organized Panic**).
    - Try again, using a different strategy (or repeating the same strategy) (**Retrying**).
How not to do it

(From an Ada textbook)

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

\[
\begin{align*}
\text{begin} \\
\text{if x < 0.0 then} \\
\text{raise Negative;} \\
\text{else} \\
\text{normal_square_root_computation;} \\
\text{end} \\
\text{exception} \\
\text{when Negative =>} \\
\text{put ("Negative argument");} \\
\text{return;} \\
\text{when others => ...} \\
\text{end; -- sqrt}
\end{align*}
\]
The call chain

Routine call

\[ r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow r_4 \]
Exception mechanism

- Two constructs:
  - A routine may contain a **rescue** clause.
  - A rescue clause may contain a **retry** instruction.

- A **rescue** clause that does not execute a **retry** leads to failure of the routine (this is the organized panic case).
Max_attempts: INTEGER is 100

attempt_transmission (message: STRING) is
  -- Transmit message in at most
  -- Max_attempts attempts.
  local
  failures: INTEGER
  do
    unsafe_transmit (message)
    rescue
      failures := failures + 1
      if failures < Max_attempts then
        retry
      end
  end
end
Transmitting over an unreliable line (2)

Max_attempts: INTEGER is 100

failed: BOOLEAN

attempt_transmission (message: STRING) is
  -- Try to transmit message;
  -- if impossible in at most Max_attempts
  -- attempts, set failed to true.
  local
  failures: INTEGER
  do
    if failures < Max_attempts then
      unsafe_transmit (message)
    else
      failed := True
    end
  end
  rescue
    failures := failures + 1
  retry
  end
Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

\[
\begin{align*}
  f (...) & \text{ is } \\
  & \text{ do } \\
  & \text{ ... } \\
  & \text{ end }
\end{align*}
\]

is an abbreviation for

\[
\begin{align*}
  f (...) & \text{ is } \\
  & \text{ do } \\
  & \text{ ... } \\
  & \text{ rescue } \\
  & \text{ -- Nothing here } \\
  & \text{ end }
\end{align*}
\]

(This is a provisional rule; see next.)
The correctness of a class

- \((1-n)\) For every exported routine \(r:\)
  \[
  \{\text{INV and Pre}_r\} \text{ do}_r \{\text{Post}_r \text{ and INV}\}
  \]

- \((1-m)\) For every creation procedure \(cp:\)
  \[
  \{\text{Pre}_cp\} \text{ do}_cp \{\text{Post}_cp \text{ and INV}\}
  \]
Exception correctness: A quiz

- For the normal body:

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

- For the exception clause:

\[
\{ ??? \} \text{ rescue}_r \{ ??? \}
\]
Quiz answers

- For the normal body:
  
  \{\text{INV and Pre}_r\} \text{ do}_r \{\text{Post}_r \text{ and INV}\}

- For the exception clause:
  
  \{\text{True}\} \text{ rescue}_r \{\text{INV}\}
Absence of a rescue clause is equivalent to a default rescue clause:

\[
\begin{align*}
&f (...) \ \text{is} \\
&\quad \text{do} \\
&\quad \quad \quad \quad \quad \quad \text{...} \\
&\quad \text{end} \\
\end{align*}
\]

is an abbreviation for

\[
\begin{align*}
&f (...) \ \text{is} \\
&\quad \text{do} \\
&\quad \quad \quad \quad \quad \quad \text{...} \\
&\quad \text{rescue} \\
&\quad \quad \text{default_rescue} \\
&\quad \text{end} \\
\end{align*}
\]

- The task of \textit{default_rescue} is to restore the invariant.
For finer-grain exception handling

- Use class *EXCEPTIONS* from the Kernel Library.

- Some features:
  - *exception* (code of last exception that was triggered).
  - *assertion_violation*, etc.
  - *raise* (“*exception_name*”)
Complementary material

- **OOSC2:**
  - Chapter 11: Design by Contract: building reliable software
  - Chapter 12: When the contract is broken: exception handling
End of lecture 9