Eiffel – Design by Contract


Presentation based on Eiffel Training Videos by Hal Weber and the book 'Eiffel the Language' by Bertrand Meyer

(Design by Contract – part 1 and 2, http://www.eiffel.com/developers/presentations/)
- Eiffel: Eiffel Development Framework (tm), focus: SW quality

- EDF consists of:
  - Eiffel development methodology
    (OO, CQS, DBC (tm), etc.)
  - Eiffel programming language & compiler
    (expression of analysis, design and implementation)
  - Eiffel development environments
    (EiffelStudio (tm), EiffelNVision (tm))

- Eiffel is a reuse centric method (reliable components crucial)
SW specification & metrics

- Specification: English, formal system based on mathematics

- reliability relative to specification:
  - correctness: SW does what it's supposed to do (spec)
  - robustness: behaves in acceptable fashion outside spec

- Design by Contract:
  - compile spec to run against, catch 'bugs' earlier

Design by Contract (invented by Eiffel): “A method of SW construction that designs the components of a system so that they will cooperate on the basis of precisely defined contracts based on a model of software correctness.”
class ROOT_CLASS
   -- ROOT_CLASS ~ main
create
   make
      -- creation procedure

feature -- Initialization

   make is
      -- Hello World, every program needs a ROOT_CLASS
      do
         io.put_string("Hello World")
      end
end -- class ROOT_CLASS
Classes

- Class: consists of features, instances of class ~ object

- Feature:
  - attribute
  - routine

- Command-Query-Separation:
  - query: “answering question” about instances (attributes and functions)
  - commands: computations that alter state of an instance (procedures)

- Uniform Access: memory or computational?
- procedure that updates the hour attribute in class TIME_OF_DAY (implementation: hour, min, sec:INTEGER)

```
set_hour (h: INTEGER) is
  -- Set the hour from 'h'
  require
    valid_h: 0 <= h and h <= 23
  do
    hour := h
  ensure
    hour_set: hour = h
    minute_unchanged: minute = old minute
    second_unchanged: second = old second
end
```
- DbC correctness for routines:
  - preconditions: true -> routine can work correctly
  - postconditions: true after execution, if routine worked correctly

- a routine is correct if pre- and postconditions are met

- reason for CQS:
  - only procedures change state of an object
  - reason about correctness of instance state using queries
Routines – Contract View

- Contract View: unaffected by implementation

- Contract View for routine supplying service:

  set_hour (h: INTEGER) is
   -- Set the hour from 'h'
   require
    valid_h: 0 <= h and h <= 23
   ensure
    hour_set: hour = h
    minute_unchanged: minute = old minute
    second_unchanged: second = old second
  end
“Design by Contract views the construction of a Software system as the fulfillment of many small and large contracts.”
Contract for a routine

<table>
<thead>
<tr>
<th>set_hour</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT</td>
<td><strong>sat pre</strong></td>
<td><strong>from post</strong></td>
</tr>
<tr>
<td></td>
<td>make sure h not too large nor too small</td>
<td>hour updated</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td><strong>sat post</strong></td>
<td><strong>from prec</strong></td>
</tr>
<tr>
<td></td>
<td>must set 'hour' to value passed in 'h'</td>
<td>may assume 'h' valid</td>
</tr>
</tbody>
</table>

- contracts violated:
  - by either party, not meeting obligations
- violated contract: SW is outside specification = DEFECT
Contract for a routine

- rules of execution: routine completes in (only) 1 of 2 ways:
  1) fulfills its contract
  2) routine fails to fulfill its contract
     - cause an exception

- routine suffering an exception reacts in (only) 1 of 2 ways:
  1) ensure object is in a valid, stable state (Retry)
  2) fail itself
     - exception passed on to caller

Contracts = “built-in reliability”
- Assertions in Eiffel: elements of formal specification expressing correctness conditions

- use of assertions:
  - pre- and postconditions of a routine
  - invariant clause of a class
  - check instruction
  - invariant of a loop instruction (also variants of a loop)

value of an assertion: true if every clause has value true, false if a clause has value false
Assertions on routines

- *pre* and *post*: precondition and postcondition of routine `rout`

- *old expression*: postconditions of routines only
  - `old exp` has same type as `exp`
  - `old exp` value on `rout` exit = `exp` on `rout` entry

- *strip expression*: part of an object that will not change

- “do not change fields except”:
  
  `equal (strip(a, b, ..), old strip(a, b, ..))`
Check instructions

- check whether a certain consistency condition is fulfilled.
- check instruction: a list of assertions packaged together
- check-correct:

  “routine $r$ is check-correct, if for every check instruction $c$ in $r$, any execution of $c$ (as part of an execution of $r$) satisfies all its assertions”
Class Invariants

- properties that must hold for any instance of a class
- valid at all critical times (= when observable by clients)
- observable: before and after each exported (= public) routine

- Class Invariant: assertion obtained by concatenating assertions
  - invariant of all parents
  - postconditions of any inherited function
  - assertion in classes' own invariant clause
- Class invariant guarantees:
  as soon as instance invalid, an exception occurs

- Class C consistent if it satisfies the following conditions:
  1) for every creation procedure p of C:
     \{\text{pre}_p\} \text{ do}_p \{\text{INV}_C\}
  2) for every routine r of C:
     \{\text{pre}_r \text{ AND INV}_C\} \text{ do}_r \{\text{post}_r \text{ AND INV}_C\}

\text{P, Q = assertions,}
\text{A = instruction or compound instruction}

\{\text{P}\} \text{ A } \{\text{Q}\} \text{ expresses the property that whenever A is executed in state in which P is true, the execution will terminate in a state where Q is true.}
Loop Invariants

- invariant assertion:
  - initialization ensures truth of INV
  - execution of loop body, in a state not satisfying exit condition, preserves the truth of INV
  => invariant and exit condition satisfied on loop exit

- loop variant (integer expression): guarantees termination
  - initialization: non-negative value

from .. invariant .. variant .. until .. loop .. end
Loop Correctness

routine is loop-correct if every loop it contains satisfies

- \{true\} INIT \{INV\}
- \{true\} INIT \{VAR \geq 0\}
- \{INV and then not EXIT\} BODY \{INV\}
- \{INV and then not EXIT and then (VAR = v)\} BODY
  \{0 \leq VAR < v\}

INV = loop's invariant, VAR = loop's variant, INIT = initialization, EXIT = exit condition, BODY = loop body.

- a routine is exception-correct if it:
  - executed Retry and ensures precondition and the invariant
  - executed no Retry and ensures the invariant.
Correctness of a Class

- Correctness of a class C: combination of correctness properties
  - it is consistent (creation => INV_c, pre_c+INV => INV)
  - every routine of C is
    - check-correct
    - loop-correct
    - exception-correct

- Ideally: tools that prove or disprove correctness of a class
  -> currently beyond reach

- but: environment supports run-time monitoring of assertions.
Run-time monitoring of Assertions

- **Eiffel**: various evaluation levels for assertions of Class C
  - **no**: no assertion checking of any kind
  - **require**: evaluate preconditions whenever execution of a routine of C begins (default)
  - **ensure**: also evaluate postconditions on return of routines
  - **invariant**: also evaluate class invariant on entry to and return from qualified calls to routines of C
  - **loop**: also evaluate the Variant and Invariant of every loop in C; after every iteration check that the variant has decreased while remaining non-negative;
  - **check or all**: also evaluate every check instruction, whenever reached.
Conclusion

- What Systems is Eiffel used in:
  - financial security, embedded systems
  - market pricing systems, manufacturing systems

- Eiffel about itself: “no magic but solid well-thought out technology, based on a few powerful ideas from computer science and software engineering”

- Performance: using C compiler optimization speeds up performance

- I found my bug in no time!