EECS3311 – Software Design

Design by Contract

Verifying Loops with loop variants invariants and

Document number:**P0542R5**
Date: **2018-06-08**
Audience: **Core Evolution Working Group**

Top news: Contracts adopted for C++20

Contracts (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup) was formally adopted for C++20.

Contracts allow preconditions, postconditions, and assertions to be expressed in code using a uniform syntax, with options to have different contract levels, custom violation handlers, and more.

Why contracts are a big deal, and related Rapperswil progress

In my opinion, contracts is the most impactful feature of C++20 so far, and arguably the most impactful feature we have added to C++ since C++11. That statement might surprise you, so let me elaborate why I think so.

Having first-class contracts support it is the first major "step 1" of reforming error handling in C++ and applying 30 years' worth of learnings.

```
double sqrt(double x) [[expects: x >= 0]];


void sort(vector<emp>& v) [[ensures audit: is_sorted(v)]];
    // could be expensive, check only when audit is requested
```

LOOPS

- Very different!
  - Class invariants
  - Loop invariants

Loop trouble

- Loops are needed, powerful
- But very hard to get right:
  - "off-by-one"
  - Infinite loops
  - Improper handling of borderline cases
- For example: binary search feature

BS1

```
from
        i := 1; j := n
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m
        else
                j := m
        end
end
Result := (x = t @ i)
```

BS2

```
from
        i := 1; j := n; found := false
until i = j and not found loop
        m := (i + j) // 2
        if t @ m < x then
                i := m + 1
        elseif t @ m = x then
                found := true
        else
                j := m – 1
        end
end
Result := found
```

BS3

```
from
i := 0; j := n
until i = j loop
        m := (i + j + 1) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

BS4

```
from
        i := 0; j := n + 1
until i = j loop
        m := (i + j) // 2
        if t @ m <= x then
                i := m + 1
        else
                j := m
        end
end
if i >= 1 and i <= n then
        Result := (x = t @ i)
else
        Result := false
end
```

OOSC2 page 381:

4 binary search algorithms:

Published but incorrect

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

Fast forward to 2015. After we had successfully verified Counting and Radix sort implementations in Java (J. Autom. Reasoning 53(2), 129-139) with a formal verification tool called KeY, we were looking for a new challenge. TimSort seemed to fit the bill, as it is rather complex and widely used. Unfortunately, we weren't able to prove its correctness. A closer analysis showed that this was, quite simply, because TimSort was broken and our theoretical considerations finally led us to a path towards finding the bug (interestingly, that bug appears already in the Python implementation). This blog post shows how we did it.

1.1 Reproduce TimSort bug in Java

```
git clone https://github.com/abstools/java-timsort-bug.git
cd java-timsort-bug
javac *.java
java TestTimSort 67108864
```

Expected output

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 40
at java.util.TimSort.pushRun(TimSort.java:413)
at java.util.TimSort.sort(TimSort.java:240)
at java.util.Arrays.sort(Arrays.java:1438)
at TestTimSort.main(TestTimSort.java:18)
```

The answer: controlled loops/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)

Review of Hoare Logic

- {Q} program{R}
  - Consider a program that starts executing in a state satisfying Q.
  - (a) If the program terminates, then it terminates in a state state satisfying R.
  - (b) The program terminates
- (a) is partial correctness
- (a) + (b) is total correctness.
- Separation of concerns

Assignment Axiom of Hoare Logic

- Hoare Axiom for Assignment:
  {R[x:=e]} x := e {R}
- R[x:=e] is called the weakest precondition to guarantee the postcondition R
- {?} i,j := i+1,x {i=j}
  - What is the weakest ? could be and still guarantee termination in a state satisfying the postcondition i=j?

R[x:=e]

= {Hoare axiom for assignment}

(i=j) [i,j := i+1,x]

=< substitution>

i+1 = x

Thus:

{i+1 = x}

i,j := i+1,x

{i=j}

Weakest Precondition Calculus

Hoare Axiom for Assignment: {R[x:=e]} x := e {R}

{Q} x:=e {R} ≡ Q ➔ wp("x := e", R)

where wp("x := e", R) ≡ R[x:=e]

i.e. the actual precondition Q must be stronger than the weakest precondition {R[x:=e]}

New and Old values

- There are different notations for the value of a program variable x in the pre-state and post-state. One possibility is a ghost variable X_0.

$\{x=X_0 \wedge ?\}\ x := x+1\ \{x > X_0\}$

$wp(\text{"}x:=x+1\text{"}, x > X_0)$
$= (x > X_0)\ [x:=x+1]$
$= x+1 > X_0$
$= x > X_0-1$
$= \{x = X_0$ in the initial state$\}$
$x > x-1$
$=$ True

Any precondition is ok
False is not ok

$\{?\}\ x := x+1\ \{x<23 \wedge y=2\}$

$wp(\text{"}x:=x+1\text{"}, x<23 \wedge y=2)$
$= (x<23 \wedge y=2)\ [x:=x+1]$
$= x+1 < 23 \wedge y=2$
$= x < 22 \wedge y=2$

Any precondition stronger than $x < 22 \wedge y=2$ is ok

- {?} x := x*x {$(x)^4=10$}

wp("x:=x*x", $(x)^4=10$)

= ($(x)^4=10$)[x:=x*x]

= $(x*x)^4=10$

= $(x)^8=10$

Alternation

{Q}
if
B
then
S1
else
S2
end
{R}

Alternation

{Q}
if
    B
then
    {Q **and** B} S1 {R}
else
    {Q **and not** B} S2 {R}
end
{R}

- We have seen how to predict and verify assignments and alternations
- What about loops?

Max1 Algorithm

What happens if a is the empty array?

```
max_in_array1 (a: ARRAY [REAL]): REAL
    local
        i: INTEGER
    do
        from
            i := a.lower
            Result := a[a.lower]
        until
            i = a.upper
        loop
            i := i + 1
            Result := Result.max (a [i])
        end
    ensure
        -- ∀j∈ a.lower..a.upper: Result ≥ a[j])
        one: across a.lower |..| a.upper as j all
                Result >= a[j.item]
            end
        two: a.has (Result)
    end
```

Predicate logic in comments

INTEGER_INTERVAL

Precondition failure on empty array

FAILED (1 failed & 3 passed out of 4)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	3	4
All Cases	3	4
State	**Contract Violation**	**Test Name**
Test1	APPLICATION	
PASSED	NONE	t1: max1 routine with array 2.1, 3.8, 2.9
FAILED	Precondition violated.	t2: max1 routine with empty array
PASSED	NONE	t3: max2 routine with array 2.1, 3.8, 2.9
PASSED	NONE	t4: max2 routine with empty array

Max1: Empty Array a

There is no index: a.lower

valid_index: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
item	ARRAY	ARRAY	3
max_in_arr...	MAX_ARRAY	MAX_ARRAY	2
t2	APPLICATION	APPLICATION	4
fast_item	PREDICATE	FUNCTION	0
item	PREDICATE	FUNCTION	5
run	ES_BOOLEA...	ES_BOOLEA...	3
run_es_test	APPLICATION	ES_TEST	22
run_espec	APPLICATION	ES_TESTABLE	2
make	APPLICATION	APPLICATION	6

Flat view of feature `max_in_array1' of class MAX_ARRAY

```
max_in_array1 (a: ARRAY [REA  ]): REAL_32
    local
        i: INTEGER_32
    do
        from
            i := a.lower
            Result := a [a.lower]
        until
            i = a.upper
        loop
            i := i + 1
```

Feature | Class

Objects {MAX_ARRAY}.max_

Name	Value	Type	Address
Exception raised	valid_index: PRECON...		
Current object	<0x103298448>	MAX_ARRAY	0x103298448
Arguments			
a	<0x103298438>	ARRAY [REAL_32]	0x103298438
Locals			
i	1	INTEGER_32	
j	Void	NONE	Void
value #3	False	BOOLEAN	
Result			
Result	0	REAL_32	

How should we deal with an empty array?

- Precondition (demanding)
- Tolerant

Max1: Another Problem

Precondition

```
max_in_array1 (a: ARRAY [REAL]): REAL
    require
        not a.is_empty
    local
        i: INTEGER
    do
        from
            i := a.lower
            Result := a[a.lower]
        until
            i = a.upper
        loop
            Result := Result.max (a [i])
        end
    ensure
        -- ∀j∈ a.lower..a.upper: Result ≥ a[j])
        one: across a.lower |..| a.upper as j all
                Result >= a[j.item]
            end
        two: a.has (Result)
    end
```

Infinite Loop

```
max_in_array2 (a: ARRAY [REAL]): REAL
    require True
```

Void safety

```
t5: BOOLEAN
    local
        a: detachable ARRAY[REAL]
        ma: MAX_ARRAY
        m: REAL
    do
        comment("t4: max2 routine with empty array")
        create ma
        m := ma.max_in_array2 (a)
        Result := m = m.negative_infinity
```

Description

VUAR(2): Non-compatible actual argument in feature call.

Error code: VUAR(2)

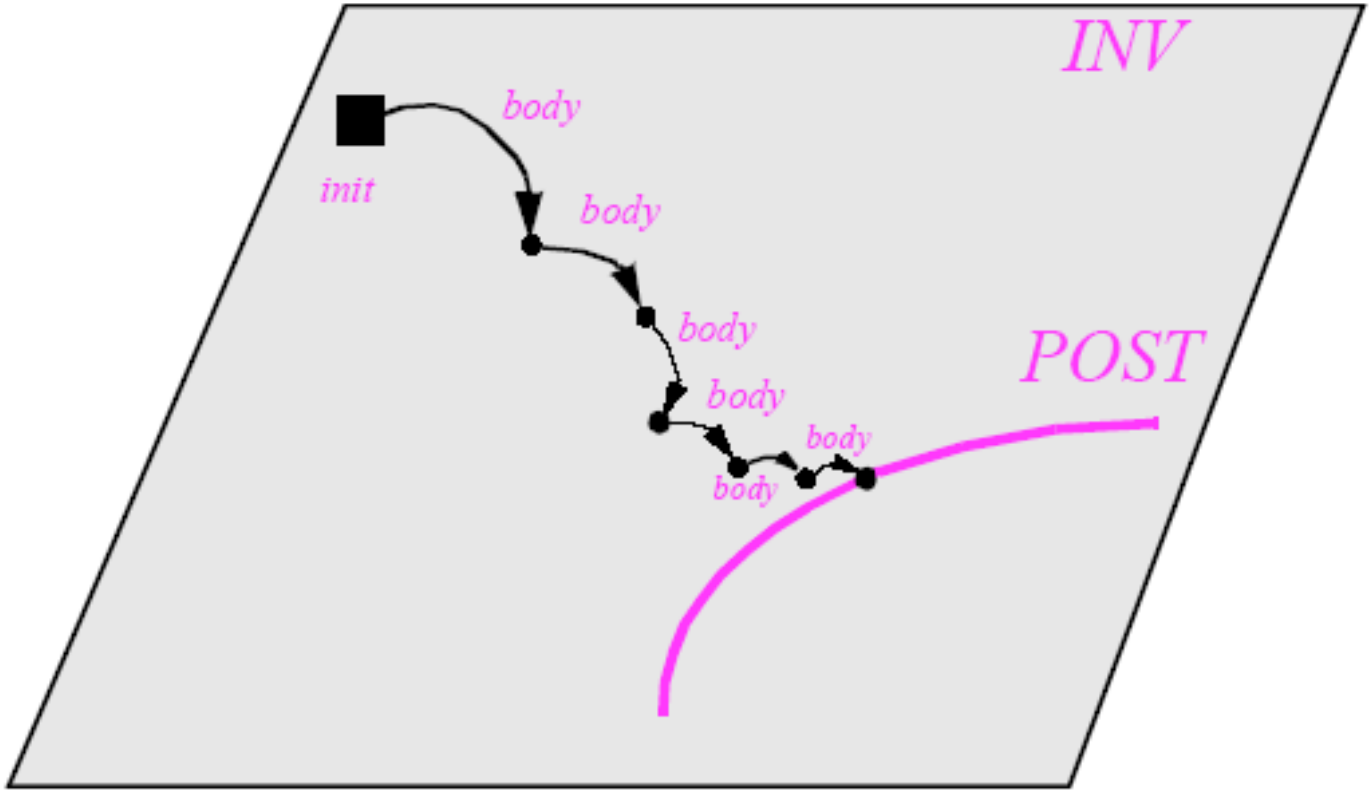
Type error: non-compatible actual argument in feature call.
What to do: make sure that type of actual argument is compatible with the type of corresponding formal argument.

Class: APPLICATION
Feature: t5
Called feature: max_in_array2 (a: ARRAY [REAL_32]): REAL_32 from MAX_ARRAY
Argument name: a
Argument position: 1
Formal argument type: ARRAY [REAL_32]
Actual argument type: **detachable** ARRAY [REAL_32]
Line: 91

```
        create ma
->      m := ma.max_in_array2 (a)
        Result := m = m.negative_infinity
```

Compile Time Error

Succesive approximations



A loop computation (from [M 1990])

{Q}
from
init
until
B
loop
body
end

{R}

{Q}
from
    {Q} init {I}
until
    B
loop
    {I **and not** B} body {I}
end

{R}

{Q}
from
    {Q} init {I}
until
    B
loop
    {I **and not** B} body {I}
end
{I **and** B}
{R}

```
max_in_array2 (a: ARRAY [REAL]): REAL
    require True
    local
        i: INTEGER
    do
        from
            i := a.lower - 1
            Result := Result.negative_infinity
        invariant
            a.lower-1 <= i and i <= a.upper
            -- ∀j∈ a.lower..i: Result ≥ a[j])
            across a.lower |..| i as j all
                Result >= a[j.item]
            end
            a.has (Result)
        until
            i = a.upper
        loop
            i := i + 1
            check
                a.lower <= i and i <= a.upper
            end
            Result := Result.max (a [i])
        variant
            a.upper - i
        end
    ensure
        -- ∀j∈ a.lower..a.upper: Result ≥ a[j])
        across a.lower |..| a.upper as j all
                Result >= a[j.item]
            end
        a.has (Result)
    end
```

Loop Invariant

Variant: must decrease with each execution of the loop

LOOP_INVARIANT_VIOLATION raised

In Feature	In Class	From Class	@
max_in_arr...	MAX_ARRAY	MAX_ARRAY	9
t0	APPLICATION	APPLICATION	4
fast_item	PREDICATE	FUNCTION	0
item	PREDICATE	FUNCTION	5
run	ES_BOOLEA...	ES_BOOLEA...	3
run_es_test	APPLICATION	ES_TEST	22
run_espec	APPLICATION	ES_TESTABLE	2
make	APPLICATION	APPLICATION	7

Flat view of feature `max_in_array3' of class MAX_ARRAY

```
max_in_array3 (a: ARRAY [REAL_32]): REAL_32
    require
        True
    local
        i: INTEGER_32
    do
        from
            i := a.lower - 1
            Result := Result.negative_infinity
        invariant
            a.lower - 1 <= i and i <= a.upper
            across
                a.lower |..| i as j
            all
                Result >= a [j.item]
            end
            a.has (Result)
```

Loop Invariant violated

Correct loop invariant?

```
max_in_array2 (a: ARRAY [REAL]): REAL
    require True
    local
        i: INTEGER
    do
        from
            i := a.lower - 1
            Result := Result.negative_infinity
        invariant
            a.lower-1 <= i and i <= a.upper
            -- ∀j∈ a.lower..i: Result ≥ a[j])
            across a.lower |..| i as j all
                Result >= a[j.item]
            end
            i >= a.lower implies a.has (Result)
            i < a.lower implies Result = Result.negative_infinity
        until
            i = a.upper
        loop
```

New Loop invariant

Variant Violation

VARIANT_VIOLATION raised

In Feature	In Class	From Class	@
max_in_arr...	MAX_ARRAY	MAX_ARRAY	11
t0	APPLICATION	APPLICATION	4
fast_item	PREDICATE	FUNCTION	0
item	PREDICATE	FUNCTION	5
		LEA...	3
		T	22
		TABLE	2
		TION	7

Flat view of feature `max_in_array3' of class MAX_ARRAY

```
max_in_array3 (a: ARRAY [REAL_32]): REAL_32
    require
        True
    local
        i: INTEGER_32
    do
        from
            i := a.lower - 1
            Result := Result.negative_infinity
        invariant
            a.lower - 1 <= i and i <= a.upper
            across
                a.lower |..| i as j
            all
                Result >= a [j.item]
            end
            i >= a.lower implies a.has (Result)
            i < a.lower im
        variant
            a.upper - i
        until
            i = a.upper
        loop
            if i < 1 then
                i := i + 1
            end
            Result := Result.max (a [i])
        end
```

i=0

i=1

Name	Value	Type
Exception raised	VARIANT_VIOLATIO...	
Current object	<0x111CC...B88>	MAX_ARRAY
Arguments		
Locals		
i	1	INTEGER_32

Variant Violation

Flat view of feature `max_in_array3' of class MAX_ARRAY

```
max_in_array3 (a: ARRAY [REAL_32]): REAL_32
    require
            True
    local
        i: INTEGER_32
    do
        from
            i := a.lower - 1
            Result := Result.negative_infinity
        invariant
                a.lower - 1 <= i and i <= a.upper
                across
                    a.lower |..| i as j
                all
                    Result >= a [j.item]
                end
                i >= a.lower implies a.has (Result)
                i < a.lower implies Result = Result.negative_infinity
        variant
                a.upper - i
        until
            i = a.upper
        loop
            if i < 1 then
                i := i + 1
            end
            Result := Result.max (a [i])
        end
```

In each iteration of the loop the variant **V** must be reduced, i.e. $\mathbf{V < V_0}$

First iteration ✓

V_0 = a.lower-i = 1 - 0 = 1

V = a.lower-i = 1 - 1 = 0

Second iteration

V_0 = a.lower-i = 1 - 1 = 0

V = a.lower-i = 1 - 1 = 0

Correct Version

```
12      max_in_array2 (a: ARRAY [REAL]): REAL
13          require True
14          local
15              i: INTEGER
16          do
17              from
18                  i := a.lower - 1
19                  Result := Result.negative_infinity
20              invariant
21                  a.lower-1 <= i and i <= a.upper
22                  -- ∀j∈ a.lower..i: Result ≥ a[j])
23                  across a.lower |..| i as j all
24                      Result >= a[j.item]
25                  end
26                  i >= a.lower implies a.has (Result)
27                  i < a.lower implies Result = Result.negative_infinity
28              until
29                  i = a.upper
30              loop
31                  i := i + 1
32                  check
33                      a.lower <= i and i <= a.upper
34                  end
35                  Result := Result.max (a [i])
36              variant
37                  a.upper - i
38              end
39          ensure
40              -- ∀j∈ a.lower..a.upper: Result ≥ a[j])
41              across a.lower |..| a.upper as j all
42                      Result >= a[j.item]
43                  end
44              a.lower <= a.upper implies a.has (Result)
45              a.lower > a.upper implies Result = Result.negative_infinity
46          end
```

Terminology

```
routine is
    require P
    do
        from initial
        invariant I
        until B
        do
            loop-body
        variant V
        end
    ensure R
    end
```

Verifying a routine with a loop

- Partial correctness
  - Show that invariant I is true before execution of the loop begins, i.e. show: {P} *initial* {I}
  - Show that: {I **and not** B} *loop-body* {I}
  - Show that: (I **and** B) **implies** R
- Termination
  - (I **and not** B) **implies** (V >= 0)
  - Show that:

    {I **and not** B **and** V = V_0}
    loop-body
    {V < V_0}
  - Note: This also means that:

    V < 0 **implies** (**not** I) **or** B
    (i.e. we have terminated)

Another version

```
max_in_array (a: ARRAY [REAL]): REAL
        require not a.is_empty
        local
                i: INTEGER
        do
                from
                        i := a.lower - 1
                        Result := Result.negative_infinity
                invariant
                        a.lower-1 <= i and i <= a.upper
                        Result = (↑ j | a.lower ≤ j ≤ i • a [j])
                until
                        i = a.upper
                loop
                        i := i + 1
                        check
                                a.lower <= i and i <= a.upper
                        end
                        Result := Result.max (a [i])
                variant
                        a.upper - i
                end
        ensure
                Result = (↑ j | a.lower ≤ j ≤ a.upper • a [j])
        end
```

Important Exercise

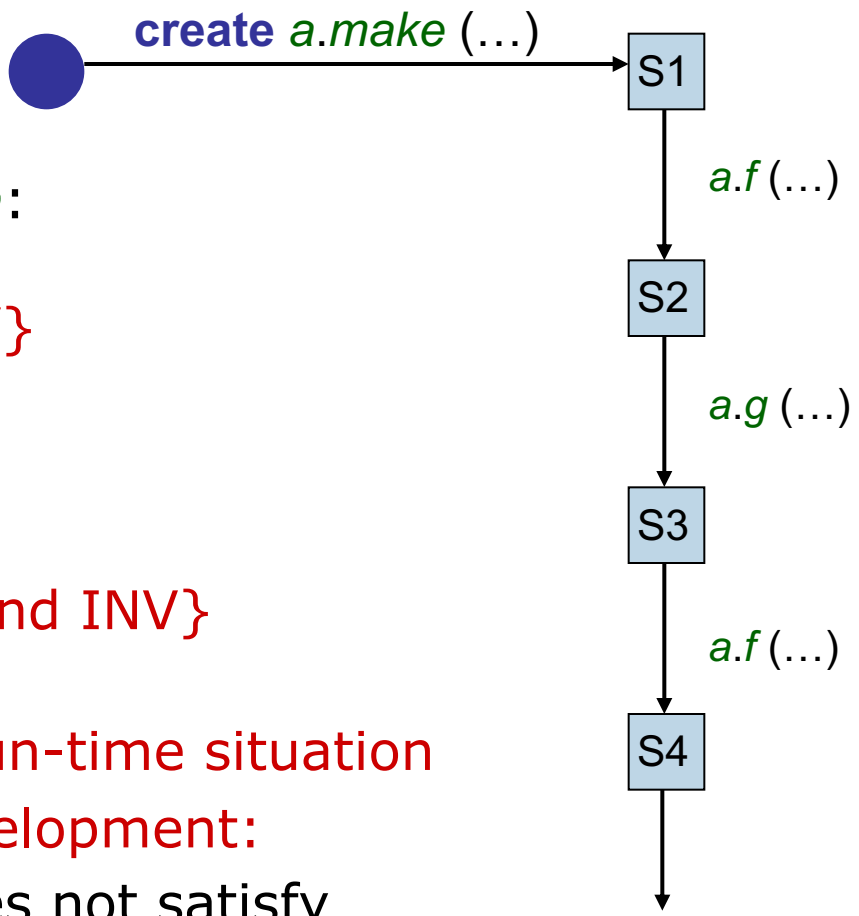
- Use agents to develop a *max* quantifier
- Use Hoare Logic to verify the total correctness of the max2 loop

Loop variants and invariants

- Syntax:

```
from
        init
invariant
        inv      -- Partial Correctness
until
        exit
loop
        body
variant
        var      -- Loop termination.
end
```

The correctness of a class



- For every creation procedure *cp*:

  $\{pre_{cp}\}\ do_{cp}\ \{post_{cp}\ \text{and INV}\}$

- For every exported routine *r*:

  $\{\text{INV and } pre_{r}\}\ do_{r}\ \{post_{r}\ \text{and INV}\}$

- 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.

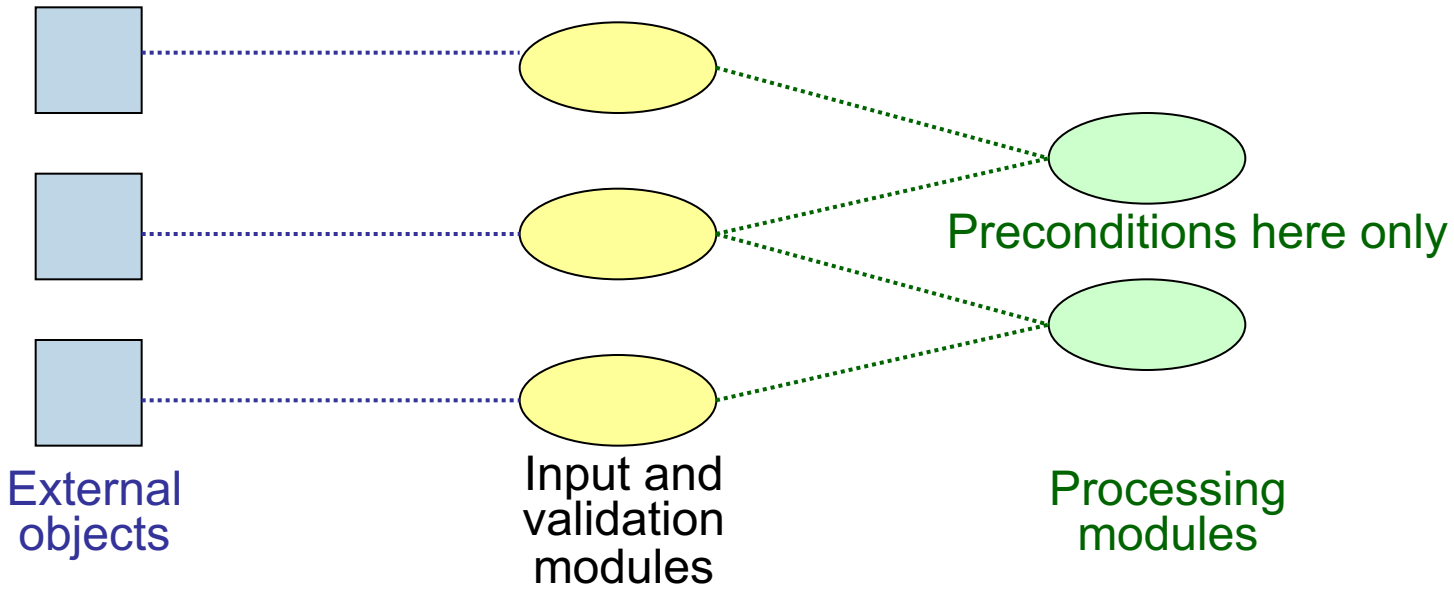
Contracts and quality assurance

- Precondition violation: Bug in the client.
- Postcondition violation: Bug in the supplier.
- Invariant violation: Bug in the supplier.

$\{P\}\ A\ \{Q\}$

Methodological notes

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



Java and assertions

- OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.
- Gosling (May 1999):
  - *“The number one thing people have been asking for is an assertion mechanism. Of course, that [request] is all over the map: There are people who just want a compile-time switch. There are people who ... want something that's more analyzable. Then there are people who want a full-blown Eiffel kind of thing. We're probably going to start up a study group on the Java platform community process.”*

(*http://www.javaworld.com/javaworld/javaone99/j1-99-gosling.html*)

- “Assert” instruction added circa Java 1.4.
- By then it was too late to add full blown DbC

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.

How to use Specifications In Java

- Use specification tests in Junit (powerful)
- Start with a high-level Eiffel model (e.g. Ehealth) of the core mission critical business logic and use the method's seamlessness to describe and validate the system specification:
  - then implement the specification in Java while ensuring that you retain preconditions, postconditions and invariants as comments and Java assertions
  - Use Java assertions everywhere! (powerful)
  - Note that methods such as TLA+ and VDM have bigger semantic gaps going from specifications to implementations

Document number:**P0542R5**
Date: **2018-06-08**
Audience: **Core Evolution Working Group**

Top news: Contracts adopted for C++20

Contracts (Gabriel Dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, Bjarne Stroustrup) was formally adopted for C++20.

Contracts allow preconditions, postconditions, and assertions to be expressed in code using a uniform syntax, with options to have different contract levels, custom violation handlers, and more.

Why contracts are a big deal, and related Rapperswil progress

In my opinion, contracts is the most impactful feature of C++20 so far, and arguably the most impactful feature we have added to C++ since C++11. That statement might surprise you, so let me elaborate why I think so.

Having first-class contracts support it is the first major "step 1" of reforming error handling in C++ and applying 30 years' worth of learnings.

```
double sqrt(double x) [[expects: x >= 0]];


void sort(vector<emp>& v) [[ensures audit: is_sorted(v)]];
    // could be expensive, check only when audit is requested
```

Lampsort in more detail

- See https://bertrandmeyer.com/2014/12/07/lampsort/

In support of his view of software methodology, Leslie Lamport likes to use the example of non-recursive Quicksort. Independently of the methodological arguments, his version of the algorithm should be better known. In fact, if I were teaching "data structures and algorithms" I would consider introducing it first.

As far as I know he has not written down his version in an article, but he has presented it in lectures; see [1]. His trick is to ask the audience to give a non-recursive version of Quicksort, and of course everyone starts trying to remove the recursion, for example by making the stack explicit or looking for invertible functions in calls. But his point is that recursion is not at all fundamental in Quicksort. The recursive version is a specific implementation of a more general idea.

Lampsort

- See https://bertrandmeyer.com/2014/12/07/lampsort/

The traditional recursive version of quicksort is:

```
sort_recursive_slice (i, j: INTEGER_32)
    require
        a.lower <= i
        i <= j + 1
        j <= a.upper
    do
        if i < j then
            partition (i, j)
            sort_recursive_slice (i, p)
            sort_recursive_slice (p + 1, j)
        end
    end
```

Lamport

Problem: Write a non-recursive version of Quicksort.

Almost no one can do it in 10 minutes.

They try to “compile” the recursive version.

For Eiffel version using Mathmodels SET[INTERVAL], see http://www.eecs.yorku.ca/research/techreports/2017/?abstract=EECS-2017-02

Lampsort in Mathmodels review

```
a: ARRAY [G]     -- array to be sorted
pivot: INTEGER   -- pivot
intervals: SET [INTEGER_INTERVAL]

lampsort
  from
    create intervals.make(a.lower |..| a.upper)
  invariant
      -- a is a concatenation of disjoint slices …
  until
    intervals.is_empty
  loop
    picked := intervals.item  -- arbitrary interval
    if picked.count > 1 then
      partition(picked.lower |..| picked.upper)
      intervals.extend(picked.lower |..| pivot)
      intervals.extend(pivot+1 |..| picked.upper)
    end
      intervals.remove(picked)
  end
```

```
partition (m, n: INTEGER_32)
        -- sets pivot index and partitions array a around it
    require a.lower <= m and m <= n and n <= a.upper
    local
        x: G; q: INTEGER_32
    do
        from
          x := a [m]; q := m + 1; p := n
        invariant
          m < q and q <= p + 1 and p <= n
          across (m + 1) |..| (q - 1) as it all
            a [it.item] <= x
          end
          across (p + 1) |..| n as it all
            a [it.item] > x
          end
        until
          q > p
        loop
          if a [q] <= x then
              q := q + 1
          elseif a [p] > x then
              p := p - 1
          elseif a [q] > x and x >= a [p] then
              swap (q, p)
              q := q + 1
              p := p - 1
          else check False end
          end
        variant p - q + 1
        end
        swap (m, p)
    ensure
      permutation (a, old a.deep_twin)
      pivot: a [p] = old a [m].deep_twin
             -- a[m..p-1] ≤ a[p]
      across m |..| (p - 1) as it all
            a [it.item] <= a [p]
      end
      -- a[p+1..n] > a[p]
      across (p + 1) |..| n as it all
          a [it.item] > a [p]
      end
    end
```

Lamport

- Engineers draw detailed plans before a brick is laid or a nail is hammered. Programmers don't. Can this be why houses seldom collapse and programs often crash?
- Blueprints help architects ensure that what they are planning to build will work. "Working" means more than not collapsing; it means serving the required purpose. Architects and their clients use blueprints to understand what they are going to build before they start building it.
- But few programmers write even a rough sketch of what their programs will do before they start coding.

Most programmers regard anything that doesn't generate code to be a waste of time. Thinking doesn't generate code, and writing code without thinking is a recipe for bad code. Before we start to write any piece of code, we should understand what that code is supposed to do. Understanding requires thinking, and thinking is hard. In the words of the cartoonist Dick Guindon:

Writing is nature's way of letting you know how sloppy your thinking is.

Blueprints help us think clearly about what we're building.

Before writing a piece of code, we should write a blueprint.

A blueprint for software is called a specification

Many reasons have been given why specifying software is a waste of time. For example: Specs are useless because we can't generate code from them. This is like saying architects should stop drawing blueprints because they still need contractors to do the construction. Other arguments against writing specs can also be answered by applying them to blueprints.

Some programmers argue that the analogy between specs and blueprints is flawed because programs aren't like buildings. They think tearing down walls is hard but changing code is easy, so blueprints of programs aren't necessary.

Wrong! Changing code *is* hard — especially if we don't want to introduce bugs.

Mathmodels: SET[G]

The query *item* from *SET*, with the precondition **not** *is_empty*, returns an element of the set. It does not matter which element.

In accordance with the Command-Query Separation principle, calling *item* does not modify the set;

to remove the element you have to use the command *remove*. The command *extend* adds an element to the set.

SET[G]

```
chosen: BOOLEAN
        do
                Result := attached item_imp
        end
```

```
feature {SET} -- Implementation

        imp: ARRAYED_LIST [G]

        item_imp: detachable CELL [G]

        set_imp (other_imp: like imp)
                do
                        imp := other_imp
```

```
choose_item
                -- Choose an arbitrary element of the
                -- set and store it in item
        require
                        not is_empty
        do
                create item_imp.put (imp [count])
        ensure
                        has (item)
                        chosen
        end
```

```
item: G
                -- Return an arbitrary member from the set.
        require
                        not is_empty
                        chosen
        do
                check
                        attached item_imp as l_item
                then
                        Result := l_item.item
                end
        ensure
                        has (Result)
        end
```

```
ry member (i.e., 'item') from the set.
```

```
l Current.deep_twin - old item
```

- Lamport uses this example to emphasize the difference between algorithms and programs, and to criticize the undue attention being devoted to programming languages.
- Algorithms work in a notation at a higher level than programming languages
- *Programs* will be specific implementations guided in particular by efficiency considerations.
- One can derive programs from higher-level algorithms through refinement. A refinement process will remove the non-determinism of {SET}.item.

Eiffel Method

- **Seamlessnes**: The Eiffel method is almost the reverse: treating the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them.
- **No semantic gap**: Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility.

- The same kinds of ideas, such as an object-oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations.

Support Change and Reuse

- One of the principal arguments for this approach is that it supports change and reuse. If everything could be fixed from the start, maybe it could be acceptable to switch notations between specification and implementation.
- But in practice specifications change and programs change, and a seamless process relying on a single notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels.
- (This problem of change is, in my experience, the biggest obstacle to refinement-based approaches. I have never seen a convincing description of how one can accommodate specification changes in such a framework without repeating the whole process. Inheritance, by the way, addresses this matter much better.)

- The example of Lampsort in Eiffel suggests that a good language, equipped with the right abstraction mechanisms, can be effective at describing not only final implementations but also abstract algorithms.
- It does not hurt, of course, that these abstract descriptions can also be executable, at the possible price of non-optimal performance. The transformation to an optimal version can happen entirely within the same method and language.

Challenge Exercise

- Program *partition* in Eiffel with loop invariant and variant (see earlier slide for solution)
- SET[G] in the base library is broken. Try *is_equal* for sets created via *extend* in different orders
- Implement a good SET abstraction with an efficient implementation
- Show that the Lamport algorithm can be directly constructed using the Eiffel notation
- Provide loop invariant and variant