Tutorial on Exception Handling

Prepared for the NODES Winter School
and Seminar, 1 - 3 February 2012
Turku, Finland

"Preventing to fail
by preparing to fail"

Emil Sekerinski
with contributions by Tian Zhang
McMaster University, Canada

Why Programs Fail

- Specification is in error:
  - does not capture the user's intent
  - incomplete, inconsistent
- Design is in error:
  - logical error, e.g. forgotten case
  - idealized hypotheses, e.g. about integer range, available memory, processing speed
  - incorrect assumptions about other components
- Underlying machine fails:
  - incorrect compilation
  - error in library implementation
  - hardware failure

Design Error: Microsoft Zune Bug ...

from techcruch.com:

30GB Zunes all over the world fail en masse

MATT BURNS | Wednesday, December 31st, 2008 | 0 Comments

It seems that a random bug is affecting a bunch, if not every, 30GB Zunes. Real early this morning, a bunch of Zune 30s just stopped working. No official word from Redmond on this one yet but we might have a gadget Y2K going on here. Fan boards and support forums all have the same mantra saying that at 2:00 AM this morning, the Zune 30s reset on their own and doesn't fully reboot. We're sure Microsoft will get flooded with angry Zune owners as soon as the phone lines open up for the last time in 2008. More as we get it.

Update 2: The solution is … kind of weak: let your Zune run out of battery and it'll be fixed when you wake up tomorrow and charge it.

Zune.net, ZuneBoards, ZuneScene, Gizmodo

Update: Reddit adds:

… Design Error: Microsoft Zune Bug

Zune bug explained in detail

DEVIN COLDEWEY | Wednesday, December 31st, 2008 | 0 Comments

Earlier today, the sound of **thousands of Zune owners crying out in terror** made ripples across the blogosphere. The response from Microsoft is to **wait until tomorrow** and all will be well. You're probably wondering, what kind of bug fixes itself?

Well, I've got the code here and it's very simple, really; if you've taken an introductory programming class, you'll see the error right away.

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

Detected Faults

- Some errors are always detected by the underlying machine:
  - indexing an array out of bounds
  - allocating memory when none is available
  - reading a file beyond its end
- Some errors can be detected by instrumenting programs:

```
class STACK
   capacity: INTEGER
   count: INTEGER
invariant
   count <= capacity
push is ...
```

- Some faults are "unfeasible" to detect:
  - only a single pointer to an object exists
  - validity of precondition and invariant of binary search
  - termination

Responding to Detected Faults

Even with best effort, possibility of fault in a complex system remains.

S_1 ; S_2 ; S_3 ; S_4 where S_1, S_3 may detect an error
in case of error, execute T instead

- Explicit testing a priori or a posteriori:

```
if S1 possible then
   S1 ; S2 ;
   if S3 possible then
      S3 ; S4
   else T
else T
```

```
S1 ;
if S1 successful then
   S2 ; S3
   if S3 successful then
      S4
   else T
else T
```

- Dedicated exception handling:

```
try
   S1 ; S2 ; S3 ; S4      ——— body
catch
   T                      ——— handler
else T
```

Exception Handling

- no additional variables and control structures interspersed; original program structure remains visible

```
f = fopen(filename, "r");
if (f == NULL) {
   ... error
} else {
   ... read file (possibly failing)
   fclose(f);
}
```

```
try {
   f = fopen(filename, "r");
   ... read file (possibly failing)
   fclose(f);
} catch {
   ... error
}
```

- useful for rare or undesired cases

- allows for imperfections during design process supporting extension and contraction

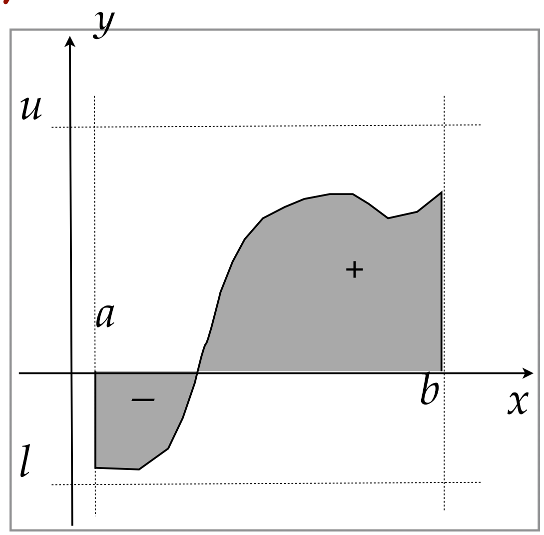
```
static void FutureFeature()
{
   // Not developed yet.
   throw new NotImplementedException();
}
```

[MS Developer Documentation for .NET]

These cases can be uniformly treated.

Example: Monte Carlo Integration in Python

- Function f evaluated randomly: may lead to arithmetic exception




```python
def area(f, a, b, l, u, n):
    c = 0
    for i in range(n):
        try:
            x = random.uniform(a, b)
            y = random.uniform(l, u)
            if 0 <= y <= f(x):
                c = c + 1
            elif f(x) <= y <= 0:
                c = c - 1
        except:
            pass
    return (u - l) * (b - a) * c / n
```

- “Rare and undesired”, but possible.
- Here exception handler does nothing, but quality of result affected.

Further Examples for Exception Handling

- Some a priory tests cannot be performed efficiently, e.g. testing arithmetic addition for possible overflow requires a subtraction, which means doubling the number of operations, e.g. in a matrix multiplication.

- A priory tests like for arithmetic overflow of floating point numbers cannot be performed reliably at all due to rounding errors.

- Errors like stack overflow on a procedure call are difficult to test for because programming languages do not offer any means.

- Transient hardware failures may occur at any time, so there is no place to test for them.

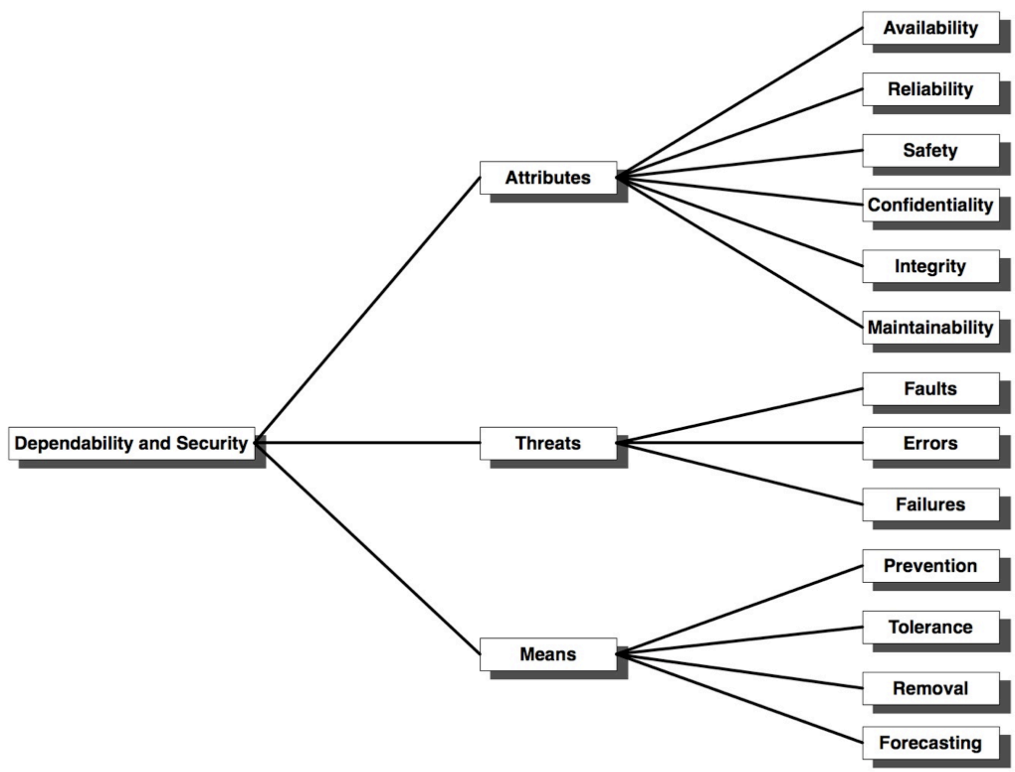
Overview

What should an exception handler do in general?

Where is an exception handler best placed?

- we give a theory based on weakest preconditions
- applicable to python, Java, C#, ...; supported in Eiffel

Where does this fit in?



Outline

- Prelude: undefinedness of expressions
- Review: weakest preconditions
- Theory: weakest exceptional preconditions
- Theory: domain properties
- Discussion: "Java vs. Eiffel" style exceptions
- Patterns: masking, propagating, flagging, rollback, degraded service, recovery block, repeated attempts, conditional retry
- Theory: total and "partial" correctness assertions
- Application: Eiffel
- (Theory: Algebraic Laws)

The Problem of Undefinedness

- If E = E is true, then is x <u>div</u> y = x <u>div</u> y also true, as in:

  b := (x div y = x div y)

- If P ∧ Q ≡ Q ∧ P is true, then are the following the same:

```
var a : array N of T ;                     ...
var n := 0 ;                               ...
while a(n) ≠ key and n < N do              while n < N and a(n) ≠ key do
    n := n + 1                                 n := n + 1
```

- Our solution is to distinguish

  terms in the logic ↔ expressions in programs

  and in particular:

  predicates (boolean terms) ↔ boolean expressions

Terms vs Expressions

- Terms in the logic, here higher-order logic:
  - used to reason about programs
  - all familiar laws hold: $P = P$ $\quad P \wedge Q \equiv Q \wedge P$ $\quad P \vee \neg P$
- Expressions in programs:
  - "look like terms", but may be undefined
  - ΔE: the definedness of E
  - 'E': the value of E
  - include conditional and, or as well as strict &, |

Definedness and Value of Expressions ...

Let c be a constant, x a variable, and assume a : array N of T:

Δc	$\equiv$	$\underline{true}$	‘c’	=	c
Δx	$\equiv$	$\underline{true}$	‘x’	=	x
$\Delta a(E)$	$\equiv$	$\Delta E \wedge 0 \le$ ‘E’ $< N$	‘a(E)’	=	a(E)
$\Delta -E$	$\equiv$	ΔE	‘−E’	=	−E
$\Delta \neg E$	$\equiv$	ΔE	‘¬E’	=	¬E
$\Delta(E \cdot F)$	$\equiv$	$\Delta E \wedge \Delta F$	‘E · F’	=	E · F
$\Delta(E\ \underline{div}\ F)$	$\equiv$	$\Delta E \wedge \Delta F \wedge$ ‘F’ $\ne 0$	‘E $\underline{div}$ F’	=	E $\underline{div}$ F
$\Delta(E\ \underline{mod}\ F)$	$\equiv$	$\Delta E \wedge \Delta F \wedge$ ‘F’ $\ne 0$	‘E $\underline{mod}$ F’	=	E $\underline{mod}$ F
$\Delta(E + F)$	$\equiv$	$\Delta E \wedge \Delta F$	‘E + F’	=	E + F
$\Delta(E - F)$	$\equiv$	$\Delta E \wedge \Delta F$	‘E − F’	=	E − F
$\Delta(E = F)$	$\equiv$	$\Delta E \wedge \Delta F$	‘E = F’	=	E = F
...					

we will leave out the ‘quotes’ as structure is preserved

With bounded arithmetic:

$$\Delta(E \cdot F) \equiv \Delta E \wedge \Delta F \wedge minint \le E \cdot F \le maxint$$

… Definedness and Value of Expressions

Let c be a constant, x a variable, and assume a : array N of T:

Δ(E and F)	≡ ΔE ∧ (E ⇒ ΔF)	'E and F'	= E ∧ F	conditional operators
Δ(E or F)	≡ ΔE ∧ (¬E ⇒ ΔF)	'E or F'	= E ∨ F	
Δ(E & F)	≡ ΔE ∧ ΔF	'E & F'	= E ∧ F	strict operators
Δ(E | F)	≡ ΔE ∧ ΔF	'E | F'	= E ∨ F	

Some laws:

¬(E and F) = ¬E or ¬F

¬(E or F) = ¬E and ¬F

	Dijkstra	Eiffel
and	cand	and then
or	cor	or else

Weakest Preconditions

$\underline{wp}(S, Q) \equiv$ weakest precondition such that S terminates with postcondition Q

Let Q be a predicate, $\underline{x}$ a list of variables, $\underline{E}$ a list of expressions, and S, T statements:

$\underline{wp}(\underline{abort}, Q)$	$\equiv$	$\underline{false}$	aborting statement
$\underline{wp}(\underline{stop}, Q)$	$\equiv$	$\underline{true}$	blocking statement
$\underline{wp}(\underline{skip}, Q)$	$\equiv$	Q	identity statement
$\underline{wp}(\underline{x} := \underline{E}, Q)$	$\equiv$	$\Delta\underline{E} \wedge Q[\underline{x} \setminus \underline{E}]$	multiple assignment
$\underline{wp}(\underline{x} :\in \underline{E}, Q)$	$\equiv$	$\Delta\underline{E} \wedge (\forall\ \underline{x}' \in \underline{E} \bullet Q[\underline{x} \setminus \underline{x}'])$	nondeterministic ass.
$\underline{wp}(S\ ;\ T, Q)$	$\equiv$	$\underline{wp}(S, \underline{wp}(T, Q))$	sequential composition
$\underline{wp}(S \sqcap T, Q)$	$\equiv$	$\underline{wp}(S, Q) \wedge \underline{wp}(T, Q)$	binary choice

Weakest Preconditions of Conditional and Iteration

Let B be boolean expression:

$$\underline{wp}(\underline{if}\ B\ \underline{then}\ S\ \underline{else}\ T, Q) \equiv \Delta B \wedge (B \Rightarrow \underline{wp}(S, Q)) \wedge (\neg B \Rightarrow \underline{wp}(T, Q))$$

Let V be an integer term and v an auxiliary variable. If

$$B \wedge P \wedge V = v \Rightarrow wp(S, P \wedge V < v)$$

$$B \wedge P \Rightarrow V > 0$$

$$P \Rightarrow \Delta B$$

P is invariant
V is variant

then:

$$P \Rightarrow wp(\underline{while}\ B\ \underline{do}\ S, \neg B \wedge P)$$

Example: Linear Search in Array

Assume a : <u>array</u> N <u>of</u> T and let:

$$P \equiv N \geq 0$$

$$S = n := 0 \; ; \; \underline{\text{while}} \; n < N \; \underline{\text{and}} \; a(n) \neq key \; \underline{\text{do}} \; n := n + 1$$

$$Q \equiv 0 \leq n \leq N \wedge (\forall\, i \mid 0 \leq i < n \bullet a(i) \neq key) \wedge (n < N \Rightarrow a(n) = key)$$

Then we can show

$$P \Rightarrow wp(S, Q)$$

using

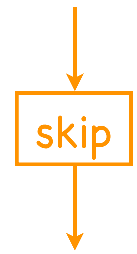
invariant: $0 \leq n \leq N \wedge (\forall\, i \mid 0 \leq i < n \bullet a(i) \neq key)$

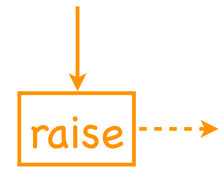
bound: $N - n$

Weakest Exceptional Preconditions ...

wp(S, Q, R) ≡ weakest precondition such that S terminates and
- on normal termination Q holds finally,
- on exceptional termination R holds finally.

Let Q, R be predicates, x a list of variables, E a list of expressions, and S, T statements:





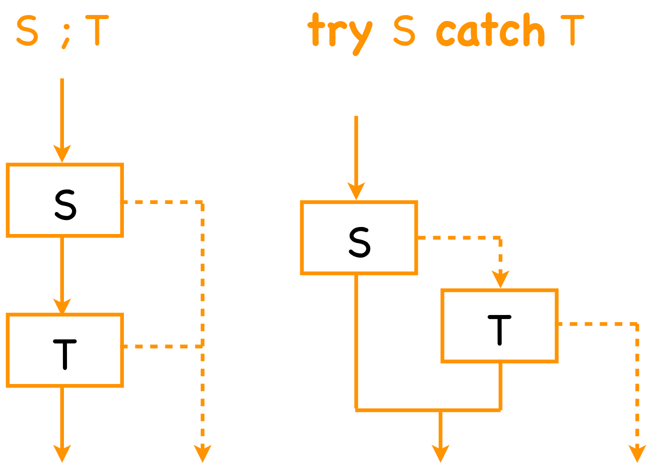
wp(abort, Q, R) ≡ false
wp(stop, Q, R) ≡ true
wp(skip, Q, R) ≡ Q
wp(raise, Q, R) ≡ R   raising exception

$wp(x := E, Q, R) \equiv (\Delta E \Rightarrow Q[x \backslash E]) \wedge (\neg \Delta E \Rightarrow R)$

$wp(x :\in E, Q, R) \equiv \Delta E \wedge (\forall x' \in E \bullet Q[x \backslash x']) \wedge (\neg \Delta E \Rightarrow R)$

$wp(S \sqcap T, Q, R) \equiv wp(S, Q, R) \wedge wp(T, Q, R)$

Weakest Exceptional Precondition of Sequential and Exceptional Compos.



<u>wp</u>(S ; T, Q, R) ≡ <u>wp</u>(S, <u>wp</u>(T, Q, R), R)

<u>wp</u>(<u>try</u> S <u>catch</u> T, Q, R) ≡ <u>wp</u>(S, Q, <u>wp</u>(T, Q, R))

exceptional composition

Weakest Exceptional Preconditions of Conditional and Iteration

$$\underline{wp}(\underline{if}\ B\ \underline{then}\ S\ \underline{else}\ T, Q, R) \equiv (\Delta B \wedge B \Rightarrow \underline{wp}(S, Q, R)) \wedge (\Delta B \wedge \neg B \Rightarrow \underline{wp}(T, Q, R)) \wedge (\neg\Delta B \Rightarrow R)$$

If

$$\Delta B \wedge B \wedge P \wedge V = v \Rightarrow wp(S, P \wedge V < v, R)$$

$$\Delta B \wedge B \wedge P \Rightarrow V > 0$$

$$\neg\Delta B \wedge P \Rightarrow R$$

P is invariant

V is variant

then:

$$P \Rightarrow wp(\underline{while}\ B\ \underline{do}\ S, \neg B \wedge P, R)$$

Properties of Weakest Exceptional Preconditions

Reduction: If S contains neither raise nor try-catch statements, then:

$\underline{wp}(S, Q) \equiv \underline{wp}(S, Q, \underline{false})$

Conjunctivity:

$\underline{wp}(S, Q, R) \wedge \underline{wp}(S, Q', R') \equiv \underline{wp}(S, Q \wedge Q', R \wedge R')$

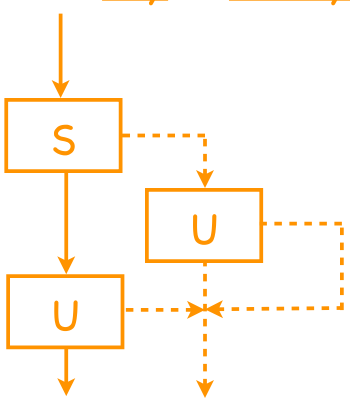
Monotonicity:

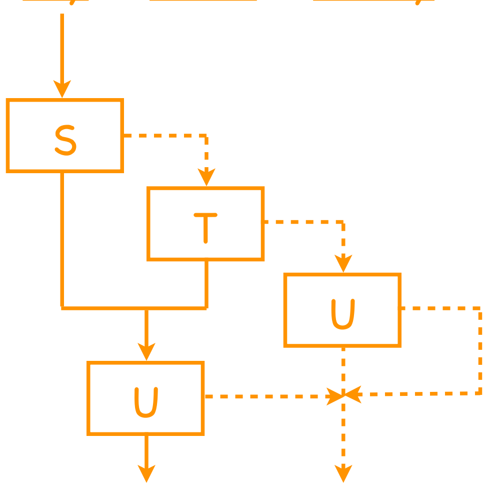
if $Q \Rightarrow Q'$ and $R \Rightarrow R'$ then $\underline{wp}(S, Q, R) \Rightarrow \underline{wp}(S, Q', R')$

Separation:

$\underline{wp}(S, \underline{true}, R) \wedge \underline{wp}(S, Q, \underline{true}) \equiv \underline{wp}(S, Q, R)$

Derived Statements





a(E) := F	=	a := a(E ← F)
if B then S	=	if B then S else skip
assert B	=	if ¬B then raise
try S finally U	=	try S catch (U ; raise) ; U
try S catch T finally U	=	try S catch try T catch (U ; raise) ; U
	=	try (try S catch T) finally U

Domains

$\underline{tr}\ S = \underline{wp}(S, \underline{true}, \underline{true})$ — termination

$\underline{nr}\ S = \underline{wp}(S, \underline{true}, \underline{false})$ — normal termination

$\underline{ex}\ S = \underline{wp}(S, \underline{false}, \underline{true})$ — exceptional termination

$\underline{en}\ S = \neg\underline{wp}(S, \underline{false}, \underline{false})$ — enabledness

Properties:

$\underline{tr}\ \underline{abort} \equiv \underline{false}$ $\quad$ $\underline{tr}\ \underline{stop} \equiv \underline{true}$ $\quad$ $\underline{tr}\ \underline{skip} \equiv \underline{true}$ $\quad$ $\underline{tr}\ \underline{raise} \equiv \underline{true}$

$\underline{nr}\ \underline{abort} \equiv \underline{false}$ $\quad$ $\underline{nr}\ \underline{stop} \equiv \underline{true}$ $\quad$ $\underline{nr}\ \underline{skip} \equiv \underline{true}$ $\quad$ $\underline{nr}\ \underline{raise} \equiv \underline{false}$

$\underline{ex}\ \underline{abort} \equiv \underline{false}$ $\quad$ $\underline{ex}\ \underline{stop} \equiv \underline{true}$ $\quad$ $\underline{ex}\ \underline{skip} \equiv \underline{false}$ $\quad$ $\underline{ex}\ \underline{raise} \equiv \underline{true}$

$\underline{en}\ \underline{abort} \equiv \underline{true}$ $\quad$ $\underline{en}\ \underline{stop} \equiv \underline{false}$ $\quad$ $\underline{en}\ \underline{skip} \equiv \underline{true}$ $\quad$ $\underline{en}\ \underline{raise} \equiv \underline{true}$

$\underline{tr}(x := E) \equiv \underline{true}$ $\quad$ $\underline{tr}(S\ ;\ T) \Rightarrow \underline{tr}\ S$ $\quad$ $\underline{tr}(S \sqcap T) \equiv \underline{tr}\ S \wedge \underline{tr}\ T$

$\underline{nr}(x := E) \equiv \Delta E$ $\quad$ $\underline{nr}(S\ ;\ T) \Rightarrow \underline{nr}\ S$ $\quad$ $\underline{nr}(S \sqcap T) \equiv \underline{nr}\ S \wedge \underline{nr}\ T$

$\underline{ex}(x := E) \equiv \neg\Delta E$ $\quad$ $\underline{ex}(S\ ;\ T) \Leftarrow \underline{ex}\ S$ $\quad$ $\underline{ex}(S \sqcap T) \equiv \underline{ex}\ S \wedge \underline{ex}\ T$

$\underline{en}(x := E) \equiv \underline{true}$ $\quad$ $\underline{en}(S\ ;\ T) \Rightarrow \underline{en}\ S$ $\quad$ $\underline{en}(S \sqcap T) \equiv \underline{en}\ S \vee \underline{en}\ T$

...

Total Correctness Assertion

$\{P\}\ S\ \{Q, R\} \equiv P \Rightarrow wp(S, Q, R)$

$\{P\}\ S\ \{Q\} \equiv P \Rightarrow \underline{wp}(S, Q, \underline{false})$

Example of annotation:

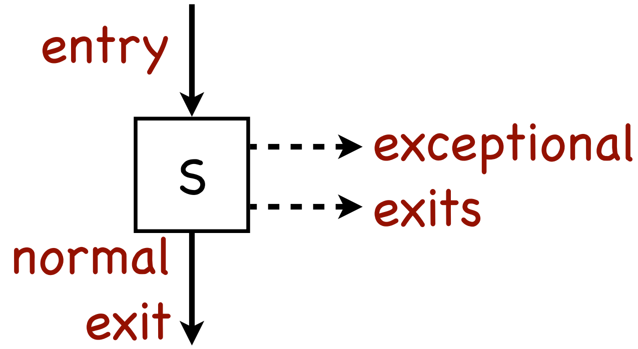
$\{P\}$	$\Leftarrow$	$\{P_1\}\ S_1\ \{Q_1, R_1\} \wedge$
<u>try</u>		$\{P_2\}\ S_2\ \{Q_2, R_2\} \wedge$
$\{P_1\}$		$(P \Rightarrow P_1) \wedge$
S_1		$(R_1 \Rightarrow P_2) \wedge$
$\{Q_1, R_1\}$		$(R_2 \Rightarrow R) \wedge$
<u>catch</u>		$(Q_1 \Rightarrow Q) \wedge$
$\{P_2\}$		$(Q_2 \Rightarrow Q)$
S_2		
$\{Q_2, R_2\}$		
$\{Q, R\}$		

Example: Saturating Vector Division

```
{true}
    i := 0
    {i = 0}
    ;
    {invariant I:
        i ∈ [0, n] ∧ ∀ j ∈ [0, i) • (b(j) ≠ 0 ∧ c(j) = a(j) div b(j)) ∨ (b(j) = 0 ∧ c(j) = maxint)}
    {variant V: n − i}
    while i < n do
        {i < n ∧ I ∧ V = v}
            try
                c(i) := a(i) div b(i)
                {i < n ∧ I ∧ b(j) ≠ 0 ∧ c(i) = a(i) div b(i) ∧ V = v, i < n ∧ I ∧ b(i) = 0 ∧ V = v}
            catch
                {i < n ∧ I ∧ b(i) = 0 ∧ V = v}
                c(i) := maxint
                {i < n ∧ I ∧ b(i) = 0 ∧ c(i) = maxint ∧ V = v}
            {i < n ∧ I ∧ ((b(i) ≠ 0 ∧ c(i) = a(i) div b(i)) ∨ (b(i) = 0 ∧ c(i) = maxint)) ∧ V = v}
            ;
            i := i + 1
        {I ∧ V < v}
    {i ≥ n ∧ I}
```

Method Specifications ...

One precondition +
one postcondition for each exit
[Cristian 84]




```
public static void search(int[] a, int x)
   throws NullPointerException, NotFoundException
/* requires: a is sorted
   ensures:  0 <= result < a.length && a[result] == x
   signals NullPointerException: a == null
   signals NotFoundException: x not in a
*/
[Liskov & Guttag 00, Leavens et al 06:JML, Barnet et al 05:Spec#]
```

All possible failures would need to be anticipated: impractical

- tools do not verify “unchecked” exceptions (Jacobs & Müller 2007)
- typical use as control structure for undesired or rare cases

… Method Specifications

In Eiffel methods have only one exceptional exit (Meyer 1997)

- specified with a precondition and a single postcondition
- exceptional exit taken if postcondition not established
- "valid" outcome even in presence of unanticipated failures

```
method is
   require
      pre
   do
      body
   ensure
      post
   rescue
      handler
```

We further elaborate on this view.

Pattern: Masking

<u>try</u> request next command
<u>catch</u> command := help

desired (but possibly weakened) postcondition is always established

If

{P} S {Q, H}
{H} T {Q}

then:

{P} <u>try</u> S <u>catch</u> T {Q}

Pattern: Masking with Re-raising

<u>try</u> process file A and output file B
<u>catch</u> (delete file B ; <u>raise</u>)

in a modular design, each module restores a consistent state before passing on the exception

If

{P} S {Q, H}
{H} U {R, R}

then:

{P} <u>try</u> S <u>catch</u> (T ; <u>raise</u>) {Q, R}

Pattern: Flagging

try (process file A and output file B ; done := true)
catch (delete file B ; done := false)

occurrence of exception is recorded for further actions

If

$\{P\}\ S\ \{Q, H\}$
$\{H\}\ T\ \{R\}$

then:

$\{P\}$
try (S ; done := true) catch (T ; done := false)
$\{(done \wedge Q) \vee (\neg done \wedge R)\}$

Pattern: Rollback with Masking

u0, v0, w0 := u, v, w ;
try display form for entering u, v, w
catch u, v, w := u0, v0, w0

prevents that an inconsistent state, e.g. broken invariant, or undesirable state, e.g. that only allows termination, is left

If

$\{P\}$ backup $\{P \wedge B\}$
$\{B\}$ restore $\{P\}$
$\{P \wedge B\}$ S $\{Q, B\}$
$\{P\}$ T $\{Q\}$

B = backup available

T can "clean up"

then:

$\{P\}$ backup ; try S catch (restore ; T) $\{Q\}$

Pattern: Rollback with Propagation

like rollback with masking, but backup is allowed to fail

If

$\{P\}$ backup $\{P \wedge B, P\}$
$\{B\}$ restore $\{P, P\}$
$\{P \wedge B\}$ S $\{Q, B\}$
$\{P\}$ T $\{Q\}$

B = backup available

T can "clean up"

then:

$\{P\}$ backup ; try S catch (restore ; raise) $\{Q, P\}$

Interlude: Partial Correctness

If {P} S {Q, P}, then S is partially correct with respect to P, Q.

Several patterns ensure partial correctness.

Eiffel method specifications can be understood as partial correctness specifications.

```
method is
   require
      pre
   do
      body
   ensure
      post
   rescue
      handler
```

Pattern: Degraded Service

```
try         -- try the simplest formula, will work most of the time
      z := √(x² + y²)
catch       -- overflow or underflow has occurred
      try
            m := max(abs(x), abs(y)) ;
            try         -- try the formula with scaling
                  t := √((x / m)² + (y / m)²)
            catch -- underflow has occurred
                  t := 1 ;
            z := m × t
      catch -- overflow on unscaling has occurred
            z := +∞ ;
            raise
```

several statements achieve the same goal, but one some are preferred over others; if the first one fails, we fall back to a less desirable one

If

$\{P\}\ S_1\ \{Q, H_1\}$
$\{H_1\}\ S_2\ \{Q, H_2\}$
$\{H_2\}\ S_3\ \{Q, R\}$

then:

$\{P\}$ try S_1 catch (try S_2 catch S_3) $\{Q, R\}$

Pattern: Recovery Block …

(Horning et al 1974, Randell 1975)

ensure A
by S_1
else by S_2
else by S_3
else error

acceptance test

alternatives

backup ;
try (S_1 ; assert A)
catch
    restore ;
    try (S_2 ; assert A)
        catch
            restore ;
            try (S_3 ; assert A)
            catch (restore ; raise)

… Pattern: Recovery Block

If

$\{P\}$ backup $\{P \wedge B, P\}$ $\{P \wedge B\}\ S_1\ \{Q_1 \wedge B, B\}$ $Q_1 \wedge A_1 \Rightarrow Q$

$\{B\}$ restore $\{P \wedge B\}$ $\{P \wedge B\}\ S_2\ \{Q_2 \wedge B, B\}$ $Q_2 \wedge A_2 \Rightarrow Q$

$\{P \wedge B\}\ S_3\ \{Q_3 \wedge B, B\}$ $Q_3 \wedge A_3 \Rightarrow Q$

then

```
{P}
    backup ;
    try (S1 ; assert A1)
    catch
        restore ;
        try (S2 ; assert A2)
        catch
            restore ;
            try (S3 ; check A3)
            catch (restore ; raise)
{Q, P}
```

Repeated Attempts

```
ra  =  while n > 0 do
           try (S ; n := −1)
           catch (T ; n := n − 1) ;
       if n = 0 then raise
```

If

$\{P\}\ S\ \{Q, R\}$

$\{R\}\ T\ \{P\}$

then

$\{n \geq 0 \wedge P\}\ ra\ \{Q, P\}$

Repeated Attempts with Rollback

```
rr  =  backup ;
       while n > 0 do
           try (S ; n := −1)
           catch (restore ; n := n − 1) ;
       if n = 0 then raise
```

If

$\{P\}$ backup $\{P \wedge B, P\}$

$\{B\}$ restore $\{P \wedge B\}$

$\{P \wedge B\}$ S $\{Q, B\}$

then:

$\{n \geq 0 \wedge P\}$ rr $\{Q, P\}$

Conditional Retry

```
cr  =  done := false ;
       while ¬done and B do
           try (S ; done := true)
           catch T ;
       if ¬done then raise
```

Mimics Eiffel's rescue and retry statements

Assume that S preserves $V = v$. If

$\{\Delta B \wedge B \wedge P\}\ S\ \{Q, R\}$

$\{R \wedge V = v\}\ T\ \{P \wedge V < v\}$

$\Delta B \wedge B \wedge P \Rightarrow V > 0$

then:

$\{P\}\ cr\ \{Q, P\}$

Eiffel Example: Approximate Square Root

Let $p \equiv 0 \le l < u \wedge l^2 \le n < u^2$

```
sqrt(n, l, u : INTEGER) : INTEGER
    {p}
    local
        m : INTEGER
    {rescue invariant: p}
    {rescue variant: u – l }
    do
        {loop invariant: p}
        {loop variant: u – l}
        from until u – l = 1 loop
            m := l + (u – l) // 2
            {p ∧ m = ( l + u ) // 2}
            if n < m * m then u := m else l := m end
            { p , p ∧ m = ( l + u ) // 2 ∧ n < m²}
        end
        {p ∧ u – l = 1}
        Result := l
rescue
    {p ∧ m = (l + u  // 2 ∧ n < m²}
    u := m
    {p}
    retry
    {retry: p}
end
{Result² ≤ n < (Result + 1)²}
```

Eiffel statements have 3 exits:

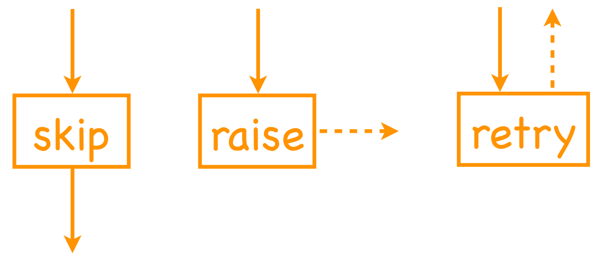
- normal exit
- raising exception
- retrying method body

The retry exit leads to a loop structure, which necessitates invariant and variant

Retry (3rd) postcondition

Eiffel Statements

$wp(skip, Q, R, S) \equiv Q$
$wp(raise, Q, R, S) \equiv R$
$wp(retry, Q, R, S) \equiv S$



Most statements are unaffected by third exit, except the rescue-loop.

Let V be an expression over the naturals. If

$\{P \wedge V = v\}\ S\ \{Q, T \wedge V = v, P \wedge V < v\}$
$\{T \wedge V = v\}\ T\ \{R, R, P \wedge V < v\}$

P is the rescue invariant
V is the rescue variant

then:

$\{P\}$ do S rescue T end $\{Q, R, S\}$

Conclusions

- Despite putting forth best effort in the design, possibility of faults remains and programs need to respond to faults.
- Exception handling with try-catch statements allows systematic treatment of faults (c.f. resumption).
- Notion of partial correctness is methodological guide: either desired postcondition is established or precondition re-established.
- Exception patterns: masking, flagging, propagating, rollback, degraded service, recovery block, repeated attempts.
- Use of exception best reserved for truly exceptional situations rather than as an extra control structure.

Outlook ...

- Some exceptions may be more severe than others, e.g. may make further attempts in the repeated attempts pattern futile: different exception types need to be distinguished.
- try-catch-finally, intuitively:
  - catch statement ensures safety by establishing a consistent state,
  - finally statement ensures liveness by freeing all resources (freeing memory; closing files, windows, network connections).
- Concurrent programs: in case of a fault in one thread/process, others may need to revert to a previous state as well. To prevent a ping-pong leading to reverting all the way to the initial state, certain checkpoints need to be established.

… Outlook

- Data abstraction and classes: class invariant has to be re-established, otherwise cascade of errors.

```
class BadStack
    public const C = 100
    private var a : array C of integer
    private var n := 0
    public method push(x : integer)
        a(n) := x ; n := n + 1              —— OK
    public method pop() : integer
        n := n - 1 ; result := a(n)         —— BAD, may break invariant 0 ≤ n ≤ C
    public method empty : boolean
        result := n = 0
    public method full : boolean
        result := n = C
```

Credit Questions

- Give three examples of programs (or fragments thereof) that you have been involved with (not from textbooks) and argue in which of the following three categories it falls. For each example, give a half-page argument why:
  - Has an appropriate use of exception handling.
  - Has an inappropriate use of exception handling.
  - Does not use exception handling, but should.
- Assuming a : array N of integer and $0 \le N \le maxint$, give the weakest precondition under which following program will not raise an exception, i.e. will not print "sum cannot be computed"; you do not have to give the proof:

```
try
    var i : integer ;
    i, sum := 0, 0 ;
    while i < N do sum, i := sum + a(i), i + 1 ;
    write(sum)
catch
    write("sum cannot be computed")
```

Assume $\Delta(E + F) \equiv \Delta E \wedge \Delta F \wedge minint \le E + F \le maxint$ and $minint < 0 < maxint$.

Further Reading

- Buhr, P. A. and Mok, W. Y. R. (2000). Advanced Exception Handling Mechanisms. IEEE Transactions on Software Engineering, 26(9), 820-836.
  Overview paper.
- Garcia, A. F., Rubira, C. M. F., Romanovsky, A., and Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. Journal of Systems and Software, 59(2), 197-222.
  Overview paper.
- Koopman, P. and DeVale, J. (2000). The Exception Handling Effectiveness of POSIX Operating Systems. IEEE Transactions on Software Engineering, 26(9), 837-848.
  Evaluates exception handling by return values.
- Liskov, B. and Guttag, J. (2000). Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
  Introduction to "Java-style" exception.

Further Reading

- Kiniry, J. (2006). Exceptions in Java and Eiffel: Two extremes in exception design and application. In Advanced Topics in Exception Handling Techniques, LNCS 4119, pages 288–300. Springer.
  Compares the two philosophies.
- Meyer, B. (1997). Object-Oriented Software Construction. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.
  Rationale for Eiffel-style exceptions.
- Sekerinski, E. and Zhang, T. (2011). Partial correctness for exception handling. In B. Bonakdarpour and T. Maibaum, editors, Proceedings of the 2nd International Workshop on Logical Aspects of Fault-Tolerance, pages 116–132.
  Applies the notion of partial correctness to some exception patterns.
- Sekerinski, E and Zhang, T. (2011). A normal form for multi-exit statements.
  Gives algebraic laws of general multi-exit statements.
- Sekerinski, E. and Zhang, T. (2012). Verification rules for exception handling in Eiffel.
  Formalizes 3-exit statements and derives verification rules.

Statement Equality ...

S = T ≡ ∀ Q, R • wp(S, Q, R) ≡ wp(T, Q, R)

Unit, left zero, associativity of ; and try-catch:

skip ; T	=	T	try raise catch T	=	T
S ; skip	=	S	try S catch raise	=	S
raise ; T	=	raise	try skip catch T	=	skip
abort ; T	=	abort	try abort catch T	=	abort
stop ; T	=	stop	try stop catch T	=	stop
(S ; T) ; U	=	S ; (T ; U)	try (try S catch T) catch U	=	try S catch (try T catch U)

Left and right distributivity of ; and try-catch over ⊓:

S ; (T ⊓ U) = (S ; T) ⊓ (S ; U)

try S catch (T ⊓ U) = (try S catch T) ⊓ (try S catch U)

(S ⊓ T) ; U = (S ; U) ⊓ (T ; U)

try (S ⊓ T) catch U = (try S catch U) ⊓ (try T catch U)

… Statement Equality …

Left distributivity of ; and try-catch over if-then-else:

(<u>if</u> B <u>then</u> S <u>else</u> T) ; U = <u>if</u> B <u>then</u> (S ; U) <u>else</u> (T ; U)

<u>try</u> (<u>if</u> B <u>then</u> S <u>else</u> T) <u>catch</u> U =
<u>if</u> B <u>then</u> (<u>try</u> S <u>catch</u> U) <u>else</u> (<u>try</u> T <u>catch</u> U) if ΔB

Merging :=, right distributivity of := over if-then-else:

x := E ; y := F(x) = x, y := E, F(E) if ΔF(E)

x := E ; <u>if</u> B(x) <u>then</u> S <u>else</u> T =
<u>if</u> B(E) <u>then</u> (x := E ; S) <u>else</u> (x := E ; T) if ΔB(E)

Left distributivity of ; over try-catch and of try-catch over ;:

(try S catch U) ; T = try (S ; T) catch (U ; T) if nr T

try (S ; U) catch T = (try S catch T) ; (try U catch T) if ex T

Shunting:

try (S ; T) catch U = S ; (try T catch U) if nr S

try S catch (T ; U) = (try S catch T) ; U if ex S

try (S ; T) catch U = (try S catch U) ; T if nr T and ex U

Merging of assert:

assert B ; assert C = assert B and C = assert C and B

here and is symmetric!

… Statement Equality

Unit and zero of finally:

try S catch T finally skip = try S catch T

try S catch raise finally U = try S finally U

try raise catch T finally U = try T finally U

Eliminating finally: useful for languages without finally

try S finally U = try (S ; U) catch (U ; raise) if nr U

try S catch T finally U = (try S catch T) ; U if nr T

try S catch (T ; raise) finally U =
try (S ; U) catch (T ; U ; raise) if nr T and nr U