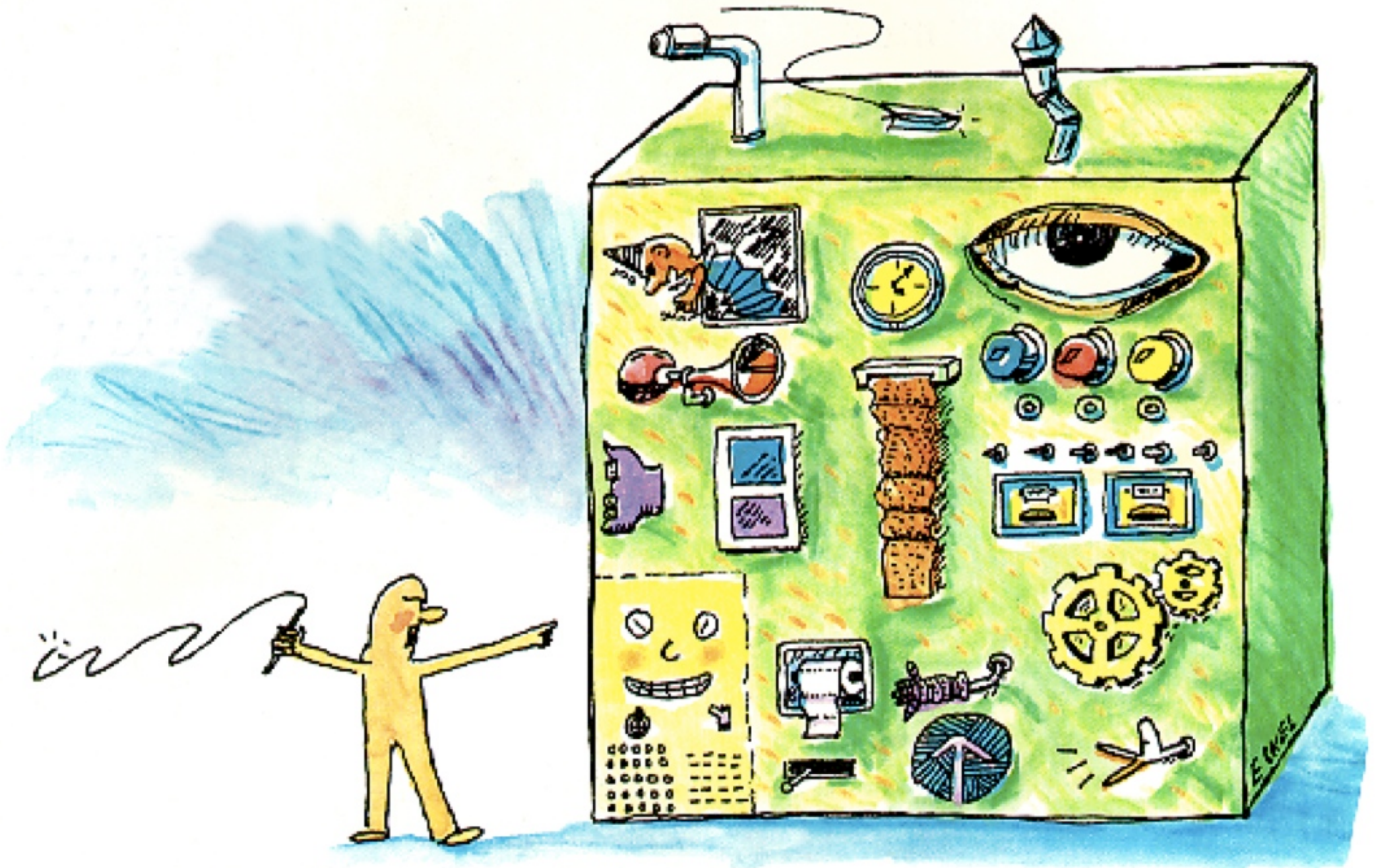
Rethinking Errors

Bruce Eckel, MindView LLC

Rethinking Errors

Bruce Eckel, Consultant
Presentation:
www.mindviewinc.com/Etc/RethinkingErrors.html
www.MindViewInc.com
www.Reinventing-Business.com
www.AtomicScala.com



Overview

- Some Error-Reporting History
- A New & Simple Approach:
  1. Clump together return object and error object
  2. Caller can’t pretend that what they get back is always good
- How it Works in Go, Scala and Python

The Problem I'm Looking At

- Incorrect inputs, not buggy function or system reliability problems
- Bad error handling produces bad programs
- Main problem is novice programmers:
  - "If the compiler or runtime doesn't tell me I'm doing anything wrong, then I must be doing everything right!"
  - Attitude can persist for a LONG time!

Error Reporting in C

- Numerous experiments, nothing worked:
  - Return an unlikely value (like -1)
  - Set a global error flag
  - Raise a signal (that you've previously set a signal handler for)
  - `setjmp`/`longjmp` (non-local goto)
- I could always pretend that I was just getting a good result

Error Reporting in C++

- Introduced exception handling (Liskov, 60's)
- Introduced exception specifications (Not enforced)
- `try`-`catch` blocks, `finally`

Error Reporting in Java

- Copied C++ Exceptions, except:
  - Enforced the exception specification via checked exceptions
  - Unified error reporting via exceptions
- Other languages:
  - Python, deeply integrated & (relatively) efficient. Even an end-of-iteration is indicated with an exception.
  - Ruby
  - Even PHP!

Exceptions Everywhere is Great!

- A single answer to all problems
- House on fire, throw an exception
- Dog piddles on carpet, throw an exception

Except …

- Overhead
  - System
  - More importantly, Programmer
- Appropriateness: One size doesn't fit all. It's not always best to
  - Panic and throw it all away
  - Jump to a different context
- Process boundaries
- Typically, only one exception can be extant at a time

Go: Initially No Exceptions

- Later added **`panic`** and **`resume`** for special cases

```
func Open(name string) (file *File, err error)

f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// Do something with the open *File f
```

Scala's "Either"

```
import com.atomicscala.AtomicTest._

def f(i:Int) =
  if(i == 0)
    Left("Divide by zero")
  else
    Right(24/i)

def test(n:Int) =
  f(n) match {
    case Left(why) => s"Failed: $why"
    case Right(result) => result
  }

test(4) is 6
test(5) is 4
test(6) is 4
test(0) is "Failed: Divide by zero"
test(24) is 1
test(25) is 0
```

Custom Error Reporting in Scala

```
import com.atomicscala.AtomicTest._
import util.{Try, Success}
import com.atomicscala.reporterr.Fail

def f(i:Int) =
  if(i == 0)
    Fail("Divide by zero")
  else
    Success(24/i)

def test(n:Int) = f(n).recover{
    case e => s"Failed: $e"
  }.get

test(4) is 6
test(5) is 4
test(6) is 4
test(0) is "Failed: Divide by zero"
test(24) is 1
test(25) is 0
```

Python: Return a Tuple as a Result

```
def f(n):
  if n > 10:
    return n * 10, False
  else:
    return None, True

def test(n):
  r, fail = f(n)
  if fail:
    print n, "failed"
  else:
    print r

test(9)
test(11)
```

- Not bad, like the Go language.

Return an Object

```
class Result(object):
  def __init__(self, result, fail=False):
    self.result = result
    self.fail = fail

class Fail(Result):
  def __init__(self, reason=True):
    Result.__init__(self, None, reason)

def f(n):
  if n > 10:
    return Result(n * 10)
  else:
    return Fail("%d is too small" % n)

def test(n):
  r = f(n)
  if r.fail:
    print r.fail
  else:
    print r.result

def test2(n):
  r = f(n)
  if isinstance(r, Fail): # Could also do this
    print r.fail
  else:
    print r.result

test(9)
test(11)
test2(9)
test2(11)
```

Slapping Bad Programmers

```
class Result(object):
  def __init__(self, result, fail=False):
    self.__result = result
    self.fail = fail

  @property
  def result(self): # Getter
    if self.__result and self.fail == False:
      return self.__result
    else:
      raise Exception(
        "Accessed Result.result during failure",
        self.fail)

class Fail(Result):
  def __init__(self, reason=True):
    Result.__init__(self, None, reason)

def f(n):
  if n > 10:
    return Result(n * 10)
  else:
    return Fail("%d is too small" % n)

def test(n):
  r = f(n)
  if r.fail:
    print r.fail
  else:
    print r.result

test(9)
test(11)
f(9).result
```

Package It Up

```
# result.py

class ResultError(Exception):
  def __init__(self, reason):
    Exception.__init__(self,
      "Accessed Result.result during failure",
      reason)

class Result(object):
  def __init__(self, result, fail=False):
    self.__result = result
    self.fail = fail

  @property # Getter
  def result(self):
    if self.__result and self.fail == False:
      return self.__result
    else:
      raise ResultError(self.fail)

class Fail(Result):
  def __init__(self, reason=True):
    Result.__init__(self, None, reason)
```

Final Version

```
from result import *

def f(n):
  if n > 10:
    return Result(n * 10)
  else:
    return Fail("%d is too small" % n)

def test(n):
  r = f(n)
  if r.fail:
    print r.fail
  else:
    print r.result

test(9)
test(11)
f(9).result
```

- Can we get rid of the if-else using functional?

```
map(print, f(n))
```

- Monads for Python: http://www.infoq.com/presentations/Monads-Code

Even C++ Is Moving In This Direction

- Go to channel9.msdn.com and look for Andrei Alexandrescu and "error"

Questions?