Concertant LLP

The Great Language Debate

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

Aims and Objectives

- Mull over what is good and bad about various languages.
- Consider the "war" between static and dynamic languages, also between functional and imperative.
  - What applications are handled well?
  - What applications are handled badly?

It is important not to try and use a language for a project where another language would clearly be better.

'Clearly better' is rarely just determined by technical programming language issues.

Hidden Agenda

Show that Python and Groovy are the dynamic languages of choice, and the languages of applications development.

Allow that C, C++, and Java have a role in computing, but not in applications development.

Worry that parallelism (multicore processors) ruin everything unless you are using a functional programming language, e.g. Erlang.

Threads are touted as the solution for multicore parallelism, but actually they are the problem.

On Expertise

Expertise is *not* the number of years of experience in a given language – though that is a factor.

Expertise is strongly related to the number of different types of language a person is fluent in.

Learning and being able to use C, C++, Java, Python, Ruby, Groovy, Fortran, Haskell, Erlang, etc. ***properly*** *makes you a better programmer.*

Subliminal Advertising

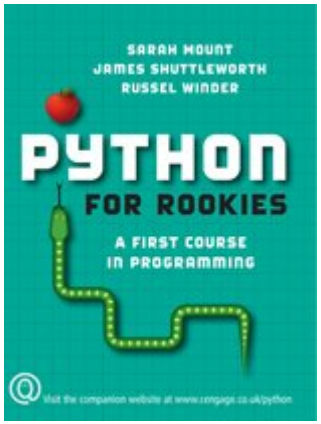
Python for Rookies

Sarah Mount, James Shuttleworth and

Russel Winder

Thomson Learning

Now called Cengage Learning.



Learners of Python need this book.

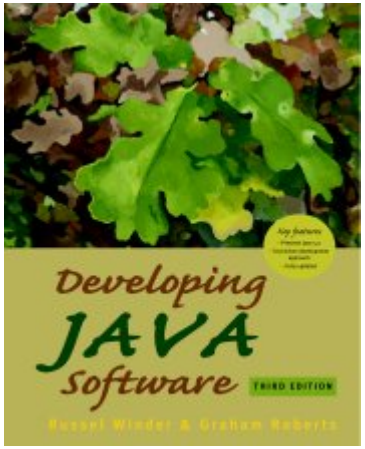
More Subliminal Advertising

Developing Java Software

Third Edition

Russel Winder and Graham Roberts

Wiley



Learners of Java need this book.

Not Advertising At All

Developing C++ Software

Second Edition

Russel Winder

Wiley

Learners of C++ used to need this book.

Programming Languages: The Players in the Game

Imperative Static:

C

C++

Fortran

Java

C#

Imperative Dynamic:

Python

Ruby

Groovy

Javascript

Lua

Scala

Perl

Functional Static:

Haskell

Erlang

Objective Caml

Scheme

Visual Basic

The Hegemony of Objects

Object-oriented programming is touted as being the pinnacle of imperative programming – C++, Java, Python, Ruby, Groovy.

These days, if you are not object-oriented, you at least have to be object based – JavaScript, C (!).

Even Perl and Fortran (!) are going this direction.

Object-oriented political correctness appears to consign functional programming to the bin.

Funccy Progging

Functional programming is often seen as too theoretical and too academic.

The absence of *side effects*, and the insistence on *referential transparency* appears to lead to problems for some applications – or more likely some programmers.

There are too few examples of big systems written in functional languages to inspire people to use them for their projects – object-oriented languages win by default.

Is Lambda Calculus of any importance to Real Programmers?

Functional Programming

The ultimate in declarative programming is to use a declarative programming language:

- Haskell
- Erlang

```
module Main where
f x = x + 1
g x = x + 2
main = do putStr ( show target ) where
  source = [ 1 , 2 , 3 ]
  target = [ f ( g x ) | x <- source ]
```

```
-module ( declarative ) .
-export ( [ start / 0 ] ) .
f ( X ) -> X + 1 .
g ( X ) -> X + 2 .
start ( ) ->
    Source = [ 1 , 2 , 3 ] ,
    Target = [ f ( g ( X ) ) || X <- Source ] ,
    io:format ( "~p~n" , [ Target ] ) ,
    init:stop ( ) .
```

Funccy Pythoneering

Python has been absorbing functional programming ideas for many years:

- Functions as first-class data.
- List comprehensions.
- Higher order functions.

`reduce` *gets too much bad press!*

Declarative is good.

Pythonic Functionalism

The programming imperative:

Programming functionally:

```
def f ( x ) : return x + 1
def g ( x ) : return x + 2
source = [ 1 , 2 , 3 ]
```

```
target = [ ]
for item in source :
    target.append ( f ( g ( item ) ) )
```

```
compose = lambda f1 , f2 : lambda x : ( f1 ( f2 ( x ) ) )
target = map ( compose ( f , g ) , source )
```

The goal is to create a new list by applying functions to the elements of an existing list.

Being Pythonic?

```
target = [ f ( g ( x ) ) for x in source ]
```

List comprehensions:
create lists declaratively, hiding the iteration.

List comprehensions come from functional languages: they are being used in more and more imperative languages.

Is this a genuine middle ground between imperative and functional?

The crucial point is the focus on the data structure and not the iteration.

Doing it Groovily

```
def f ( x ) { x + 1 }
def g ( x ) { x + 2 }
source = [ 1 , 2 , 3 ]

target = source.collect { x ->  f ( g ( x ) ) }
```

Functions as parameters.

Iteration is there but is is hidden in this declarative expression.

Collect is the same semantics as map just a different name.

Declarative is a Trend

Not only have Python and Groovy been absorbing functional ideas, so have Ruby, and even C++ !

```
std::vector<std::string> v ;
```

The imperative way: using iterators.

```
for ( std::vector<std::string>::const_iterator i = v.begin() ; i != v.end() ; ++i ) {
  puts(i->c_str()) ;
}
```

The declarative way: using generic algorithms.

```
std::copy(v.begin(), v.end(), std::ostream_iterator<std::string>(std::cout)) ;
```

The trend is away from imperative expression towards declarative expression.

The Declarative Focus

- Being declarative is about expressing the result of a computation, not how to progress the computation.
- Declarative expression admits greater optimization and use of efficient algorithms:
  - Whole array operations in Fortran.
  - STL in C++.
- The emphasis is on relationships between data.

Fortran and Abstraction

Fortran 77

```
      REAL a(5, 5), b(5, 5), c(5, 5)
      DO 20 i = 1, 5
        DO 10 j = 1, 5
          c(j, i) = a(j, i) * b(j, i)
10      CONTINUE
20    CONTINUE
```

Fortran 90

```
REAL, DIMENSION (5, 5) :: a, b, c
c = a * b
```

It's all about declarative expression

Data Structures

- Scalars
  - Every language supports these, though the semantics of variables can be quite different: value holder vs. value label.
- Sequences
  - Arrays, lists, etc. a wide range of performances depending on the detailed implementation.
- Maps
  - Aka associative arrays, hashes, dictionaries.

Clearly there are others but these are 'core' and will suffice for the argument.

Sequence Examples – C

C has arrays, you can build you own linked lists.

No array bounds checking.

Very low-level memory model.

Excellent for systems programming, useless for applications programming.

```
#include <stdio.h>
int main ( ) {
  int array [] = { 1 , 2 , 3 ,4 } ;
  int i ;
  array[2] = array[3] ;
  for ( i = 0 ; i < 4 ; ++i ) { printf ( "%i\n" , array[i] ) ; }
  return 0 ;
}
```

Sequence Examples – C++

C++ has all the array and linked list ideas of C.

C++ also has templates and the standard library which contains some high quality data structure types, cf. vector and list.

```
#include <vector>
#include <iostream>
#include <iterator>
int main ( ) {
  std::vector<int> sequence ;
  sequence.push_back ( 1 ) ;
  sequence.push_back ( 2 ) ;
  sequence.push_back ( 3 ) ;
  sequence.push_back ( 4 ) ;
  sequence[3] = sequence[2] ;
  std::copy ( sequence.begin() , sequence.end() , std::ostream_iterator<int>(std::cout) ) ;
  return 0 ;
}
```

There has to be a better way.

Declarative expression of algorithm.

Sequence Examples – C++ Alternative

```
#include <vector>
#include <iostream>
#include <iterator>
int main ( ) {
  int data [] = { 1 , 2 , 3 , 4 } ;
  std::vector<int> sequence  ( &data[0] , &data[sizeof(data)/sizeof(data[0])] ) ;
  sequence[3] = sequence[2] ;
  std::copy ( sequence.begin() , sequence.end() , std::ostream_iterator<int>(std::cout) ) ;
  return 0 ;
}
```

Arrays don't know their own length.

Declarative expression of algorithm.

Verbosity level is very high.

Sequence Examples – Java

Java Collection Framework similar to the C++ standard library data structures.

```
import java.util.Arrays ;
import java.util.ArrayList ;
import java.util.List ;
public class SequenceUse {
  public static void main ( final String[] args ) {
    final List<Integer> sequence = Arrays.asList ( 1 , 2 , 3 , 4 ) ;
    sequence.set ( 3 , sequence.get ( 2 ) ) ;
    System.out.println ( sequence ) ;
  }
}
```

Sequence Examples – Haskell

Lists are a built-in type, and it shows.

Names are single assignment and data is immutable – this also shows.

Easy to pick off the head element.

Random access is a real pain: there are arrays, but . . .

```
module Main where
main = do putStr ( show theSequence ) where
      initialSequence = array ( 0 , 3 ) [ ( 0 , 1 ) , ( 1 , 2 ) , ( 2 , 3 ) , ( 3 , 4 ) ]
      theSequence = initialSequence // [ ( 3 , initialSequence ! 2 ) ]
```

Sequence Examples – Erlang

Same issues as with Haskell: lists are fine but not random access structures.

```
-module ( sequenceUse ) .
-export ( [ start / 0 ] ) .
start ( ) ->
    TheSequence = [ 1 , 2 , 3 , 4 ] ,
    io:format ( "~p~n" , [ TheSequence ] ) ,
    init:stop ( ) .
```

Give up on doing this sensibly

Functional programming language, has a head/tail view of list.

Sequence Examples Dynamically – Python, Ruby, Groovy

```
sequence = [ 1 , 2 , 3 , 4 ]
sequence[3] = sequence[2]
print sequence
```

```
[1, 2, 3, 3]
```

```
sequence = [ 1 , 2 , 3 , 4 ]
sequence[3] = sequence[2]
puts( sequence )
```

```
1
2
3
3
```

```
sequence = [ 1 , 2 , 3 , 4 ]
sequence[3] = sequence[2]
println ( sequence )
```

```
[1, 2, 3, 3]
```

Sequences Summary

C shows it is low level.

C++ shows it is more expressive but still a bit low-level.

Functional programming languages like Haskell and Erlang have great support for a head/tail model of sequence.

Dynamic languages like Python, Ruby, and Groovy are expressive and minimalistic.

Map Examples – C

What's a map?

Map Examples – C++

```
#include <map>
#include <iostream>

int main ( ) {
  std::map<std::string,int> theMap ;
  theMap["fred"] = 2 ;
  std::cout << theMap ;
}
```

Trap for the unwary.

Map Examples – C++

```
|> g++ mapUse.cpp
mapUse.cpp: In function 'int main()':
mapUse.cpp:7: error: no match for 'operator<<' in 'std::cout << theMap'
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:67: note: candidates are: std::basic_ostream<_CharT, _Traits>&
std::basic_ostream<_CharT, _Traits>::operator<<(std::basic_ostream<_CharT, _Traits>& (*)(std::basic_ostream<_CharT, _Traits>&)) [with _CharT = char, _Traits =
std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:78: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_ios<_CharT, _Traits>& (*)(std::basic_ios<_CharT, _Traits>&)) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:90: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::ios_base& (*)(std::ios_base&)) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:241: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:264: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long unsigned int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:102: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(bool) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:125: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(short int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:157: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(short unsigned int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:183: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:215: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(unsigned int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:288: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long long int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:311: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long long unsigned int) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:361: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(double) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:335: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(float) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:384: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long double) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:407: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(const void*) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../include/c++/4.1.2/bits/ostream.tcc:430: note:  std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_streambuf<_CharT, _Traits>*) [with _CharT = char, _Traits = std::char_traits<char>]
```

Map Examples – Java

```
import java.util.HashMap ;
import java.util.Map ;

class MapUse {
  public static void main ( final String[] args ) {
    Map<String,Integer> theMap = new HashMap<String,Integer> ( ) ;
    theMap.put ( "fred" , 2 ) ;
    System.out.println ( theMap ) ;
  }
}
```

{fred=2}

Map Examples – Haskell

```
module Main where
import Data.Map
main = do putStr ( show theMap ) where
        theMap = fromList [ ( "fred" , 2 ) ]
```

```
fromList [("fred",2)]
```

Map Examples – Erlang

```
-module ( mapUse ) .
-export ( [ start / 0 ] ) .
start ( ) ->
    TheMap = { "fred" , 2 } ,
    io:format ( "~p~n" , [ TheMap ] ) .
```

{“fred”,2}

Map Examples – Python, Ruby, Groovy

```
theMap = { 'fred' : 2 }
print theMap
```

```
{'fred':2}
```

```
theMap = { 'fred' => 2 }
puts( theMap )
```

```
fred2
```

```
def theMap = [ 'fred' : 2 ]
 println ( theMap )
```

```
["fred":2]
```

Map Summary

C is low level, maps are not a memory structure they are a data structure.

C++ can handle things, well and efficiently.

Functional programming languages can have difficulties since they are head/tail list focused.

Dynamic language like Python, Ruby, and Groovy just do the right thing.

The expressiveness and lack of 'noise' means that the dynamic languages are easy to work with.

Closures – Is Python Being Left Behind?

Ruby, Groovy, even Java, and in a different way, C ++ are moving towards declarative expression.

Closures invert flow control expression.

```
def data = [ 3 , 4 , 5 ]

def doSomething = { i -> println ( i ) }

data.each doSomething
```

Ivan Moore's articles on closures in Python are well worth reading.

http://ivan.truemesh.com/archives/000392.html

Closures – Is Python Being Left Behind?

Python remains based on iteration using for and while.

Iterables do avoid indexing – it is the indexing that is the real problem.

```
data = [ 3 , 4 , 5 ]

def doSomething ( i ) : print i

for d in data :
    doSomething ( d )
```

```
for i in range ( 10 ) :
    doSomething ( data[i] )
```

Being Machine Specific

Hardware related programming, requires low-level languages:

- Assembly language is good, and bad.
- C is a portable assembler.
- C++ is generally better.

Despite being a representation of PDP-8 assembly language.

```
controller * x = 0xffffea00 ;
x->datum = datum ;
```

The Static vs. Dynamic War

- Size matters.
  - With large projects and many developers, static typing can be extremely beneficial.
  - Programs that use dynamic typing need to be understood as a whole by all developers working on the program.
  - Rapid prototyping is clearly better supported by dynamically types languages.

Is static typing a management tool for constraining the incompetence of bad programmers?

Type Inference

The functional programming languages, e.g. Haskell, Erlang, and also mixed paradigm languages like Scala use type inferencing.

Even C++ is introducing type inferencing in C++0x!

Dynamic languages cannot use this.

Can statically typed languages with type inferencing become as easy of use and expressive as the dynamic languages?

Commercial Strategies

Systems that abstract operating systems, e.g. Java, Python, Ruby, are a threat to proprietary operating system vendors.

Thus the rise of operating specific variants:

- .NET
- C#
- IronPython
- IronRuby
- J++
- J#

Is the intention of these languages and frameworks to break platform independence and create operating system lock in?

The Role of Graphics for Tie-in

Graphics is notoriously platform dependent.

Four major platforms: Windows, Mac OS X, Gnome, KDE.

Rise of the graphics adaptors: wxWidgets, Swing/AWT.

Perhaps wxPython wins over PyGTK, and Tkinter?

Platform-specific appearance vs. platform-independent appearance?

Open and Closed Classes

- Python and Ruby have open classes.
- Groovy, C++, Java have closed classes.
  - Groovy has Expandos and ExpandoMetaClass so there are features of openness.
- Is the ability to alter a class a safe thing to allow?

Code Layout – Is Python Out of Step?

Python is like the functional programming languages – whitespace matters.

Formatting and indentation assumptions mean less clutter.

Ruby and Groovy are 'curly bracket' languages where there are explicit markers and much less constraint on formatting, but they don't have clutter either.

Are Python's formatting constraints just a 'religious' issue?

Parallelism – Coping with Multicore

Fortran, C and C++ use OpenMP and MPI.

C++ is getting a threads model based on pthreads.

Java has threads. Groovy is just like Java.

All of the above are shared memory based and hence have synchronization problems.

Erlang has a message passing and so has no shared memory synchronization problems.

Python and Ruby fail to support parallelism sanely.

This slide may well be changed before the conference to include more material and hence actually be useful.

Summary

Dynamic languages (Python, Ruby, Groovy) are expressive language, but have no ability to work directly with hardware as C and C++ do.

Dynamic languages work well for all applications development, especially platform independent GUIs.

Multi-paradigm languages (C++, Python, Groovy, Scala) allow easier expression as you can use the right paradigm for a given algorithm.