# Closures & Functional Programing

They are a block of code plus the bindings to the environment they came from (Ragusa Idiom).

“Refactoring is improving the design of code after it has been written”

Kleiner
Kommunikation.....

# Agenda EKON 06.11.2012

- Closure as a Code Block
- Long History
- Dynamic Languages
- Functional Programming Examples
- Closures & Refactoring
- Case Study Experiences with a Polygraph Design
- Links

# Code Blocks

How many times have you written a piece of code and thought “I wish I could reuse this block of code”?

How many times have you refactored existing code to remove redundancies?

How many times have you written the same block of code more than once before realising that an abstraction exists?

# Why Closures?

Languages that support closures allow you to define functions with very little syntax. While this might not seem an important point, I believe it's crucial - it's the key to make it natural to use them frequently. Look at Lisp, Smalltalk, or Ruby code and you'll see closures all over the place.

UEB: demos/threads/thrddemo.exe

# What‘s a closure ?

Never touch a running system ?!:

//Calls n times a value:

- python
-
- def n_times(a_thing)
- return proc{ |n| a_thing * n}
- end

- c1 = n_times(23)
- puts c1.call(1) # -> 23
- puts c1.call(3) # -> 69

# Closure Counter

- def counter(start)
  return proc {start *= 3}
  end

k = counter(1)
g = counter(10)

- puts k.call # -> 3
- puts k.call # -> 9
- puts g.call # -> 30
- puts k.call # -> 27

UEB: 8_pas_verwechselt.txt

# Closure: Definition

- Closures are reusable blocks of code that capture the environment and can be passed around as method arguments for immediate or deferred execution.
- Simplification: Your classes are small and your methods too; you've said everything and you've removed the last piece of unnecessary code.

# Closure time

- def times(n):

```
def _f(x):
  return x * n
return _f
```

- t3 = times(3)
- print t3 #
- print t3(7) # 21

# Closure in Python

```
def generate_power_func(n):
  def nth_power(x): //closure
          return x**n
return nth_power
```

```
>>> raised_to_4 = generate_power_func(4)
```

debug:
id(raised_to_4) == 0x00C47561 == id(nth_power)).

>>> raised_to_4(2) → 16

# History

- Lisp started in 1959
- Scheme as a dialect of Lisp
  - One feature of the language was function-valued expressions, signified by *lambda*.
- Smalltalk with Blocks.
- Lisp used something called *dynamic scoping*.

# Dynamic Languages

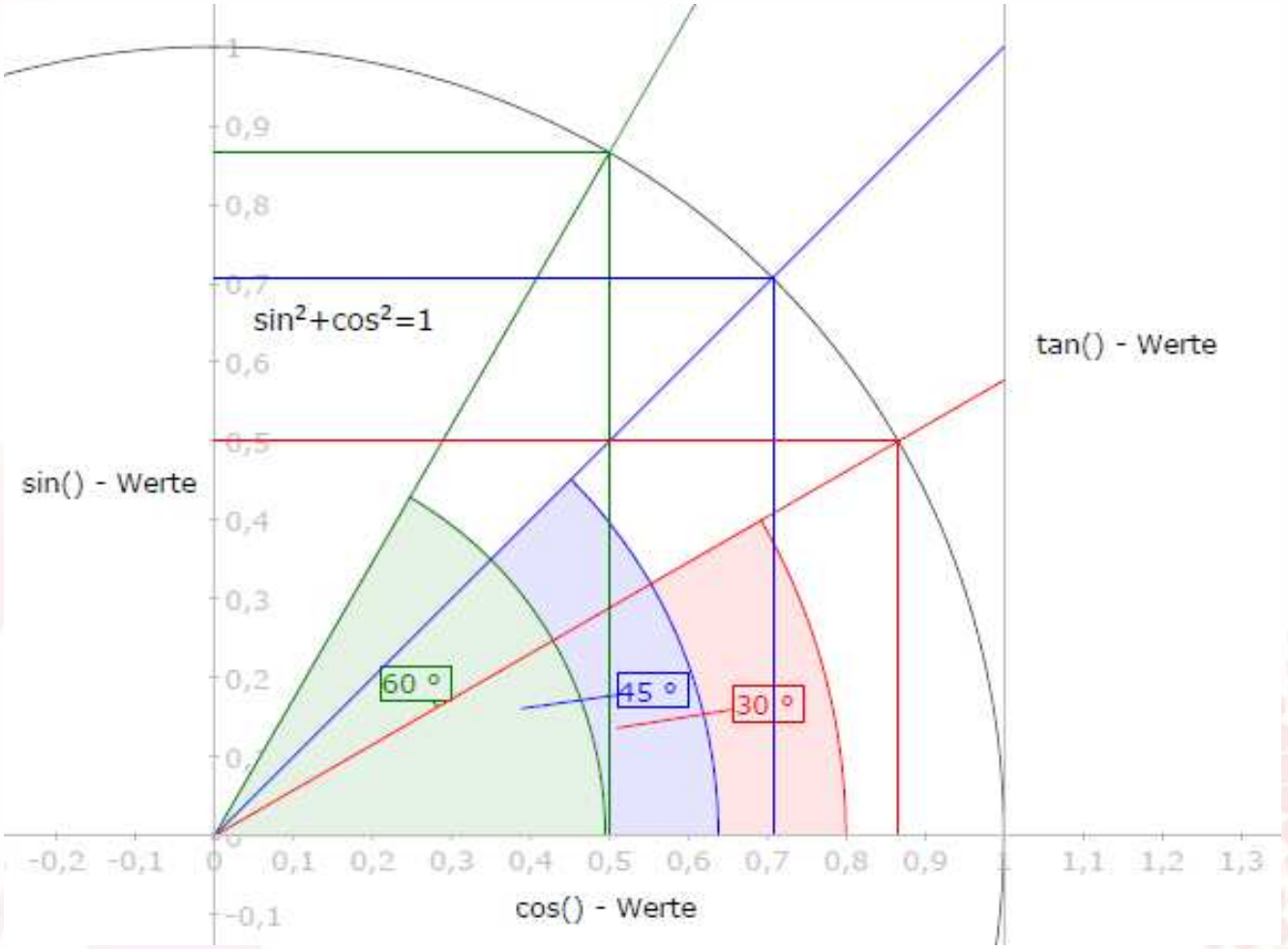
The increase in importance of web services, however, has made it possible to use dynamic languages (where types are not strictly enforced at compile time) for large scale development. Languages such as Python, Ruby, Delphi and PHP.

Scheme was the first language to introduce closures, allowing full lexical scoping, simplifying many types of programming style.

to discuss:

```
for i:= 0 to SourceTable.FieldCount - 1 do
        DestTable.Fields[i].Assign(SourceTable.Fields[i]);
  DestTable.Post;
```

# Closure in Math Optimization



# Function Pointers

- Possible Callback or Procedure Type
- Type

```
    TMath_func = Procedure(var x: float); //Proc Type

procedure fct1(var x: float); //Implement
begin
  x:= Sin(x);
end;
```

- var
- fct1x:= @fct1 //Reference
- fct_table(0.1, 0.7, 0.4, fct1x); //Function as Parameter
- fct_table(0.1, 0.7, 0.4, @fct1); //alternativ direct

# Anonymous I

```
type
  TWriter = reference to procedure;
var xt: TWriter;
begin
  xt:= procedure
  begin
     writeln('Display this Text Ext');
  end;
 xt;
readln;
end;
```

# Anonymous II

```
type
   TIntegerConvertFunc = reference to function(s: string):
integer; //Has a return type

var
  myFunction: TIntegerConvertFunc;
begin
  myfunction:= function(s: string): integer
    begin
       result:= IntToStr(s);
    end;
// ...
```

# Delegates

## Method Pointers (Object Organisation)

```
ppform:= TForm.Create(self);
 with ppform do begin
   caption:= 'LEDBOX, click to edit, dblclick write out pattern'+
                ' Press <Return> to run the Sentence';
   width:= (vx*psize)+ 10 + 300;
   height:= (vy*psize)+ 30;
   BorderStyle:= bsDialog;
   Position:= poScreenCenter;
   OnKeyPress:= @FormKeyPress
   OnClick:= @Label1Click;
   OnClose:= @closeForm;
   Show;
 end
```

UEB: 15_pas_designbycontract.txt

# Closure as Parameter

- Consider the difference

- def managers(emps)
- return emps.select {|e| e.isManager}
- end

  Select is a method defined on the Ruby collection class. It takes a block of code, a closure, as an argument.

- In C or Pascal you think as a function pointer
- In Java as an anonymous inner class
- In Delphi or C# you would consider a delegate.

# Function Conclusion

**Step to Closure Callback**

**A block of code plus the bindings to the environment they came from.**

**This is the formal thing that sets closures apart from function pointers and similar techniques.**

UEB: 14_pas_primetest.txt

# Closure as Object

• When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is called lexical scoping.

• Evaluating a closure expression produces a closure object as a reference.

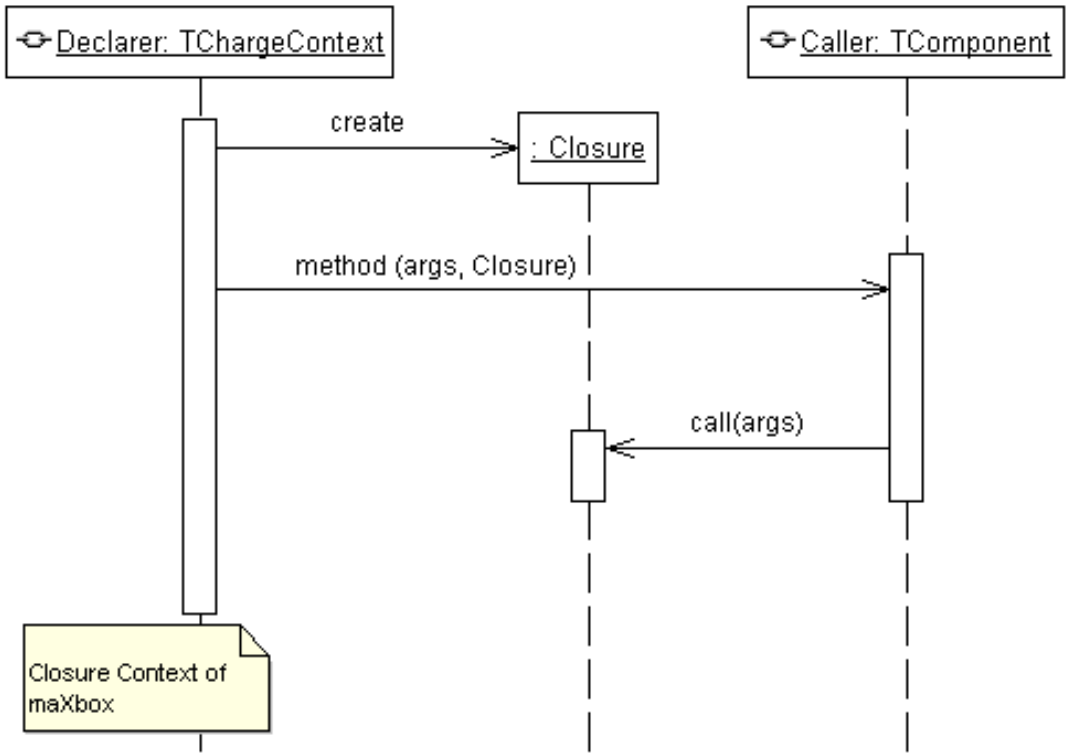
• The closure object can later be invoked, which results in execution of the body, yielding the value of the expression (if  one was present) to the invoker.

# Closure as Callback

- function AlertThisLater(message, timeout)
  {
  function fn() { alert(message); }
  window.setTimeout(fn, timeout);
  }
  AlertThisLater("Hi, JScript", 3000);

A closure is created containing the message parameter, fn is executed quite some time after the call to AlertThisLater has returned, yet fn still has access to the original content of message!

# Closure SEQ


Declarer: TChargeContext
Caller: TComponent
create
: Closure
method (args, Closure)
call(args)
Closure Context of
maXbox

# Closure as Thread

```
groovy> Thread.start { ('A'..'Z').each {sleep 100;
                                        println it} }
groovy> Thread.start { (1..26).each {sleep 100;
                                        println it} }
```

```
>>> A
>>> 1
>>> B
>>> 2
```

# Lambda Expression

- There are three main parts of a function.

1. The parameter list
2. The return type
3. The method body

- A Lambda expression is just a shorthand expression of these three elements.

```
// C#
```

- string s => Int.Parse(s)

# Closure Syntax I

```
procedure OuterMethod;
var upperScope: string;
      procedure InnerMethod;
      var innerScope: string;
      begin
            // Here, upperScope and innerScope are visible.
            // lowerScope is not visible.
            writeln(text);
      end;
var lowerScope: string;
begin
// upperScope and lowerScope are visible here.
  InnerMethod;
end;
```

# Closure Syntax II

```
procedure TMyClass.DoCount;
var j: integer;
ShowCounter: TProc; // Method no longer has a parameter.
begin
  ShowCounter:= procedure
                  begin
                  // j is known here because it is wrapped in a closure
                  // and made available to us!
                    writeln(IntToStr(j));
                  end;
      for j:= 1 to 10 do
        ShowCounter; // j is no longer passed
end;
```

# Syntax Schema

```
>>> def outer(x):
        ... def inner(y):
            ... return x+y
        ... return inner
...
```

- >>> customInner=outer(2)
- >>> customInner(3)
- result: 5

# Let‘s practice

- 1
- 11
- 21
- 1211
- 111221
- 312211
- ???

Try to find the next pattern, look for a rule or logic behind !

# Before C.

```
function runString(Vshow: string): string;
var i: byte;
Rword, tmpStr: string;
cntr, nCount: integer;
begin
cntr:=1; nCount:=0;
Rword:=''; //initialize
tmpStr:=Vshow; // input last result
for i:= 1 to length(tmpStr) do begin
  if i= length(tmpstr) then begin
     if (tmpStr[i-1]=tmpStr[i]) then cntr:= cntr +1;
     if cntr = 1 then  nCount:= cntr
     Rword:= Rword + intToStr(ncount) + tmpStr[i]
  end else
   if (tmpStr[i]=tmpStr[i+1]) then begin
      cntr:= cntr +1;
      nCount:= cntr;
   end else begin
     if cntr = 1 then cntr:=1 else cntr:=1; //reinit counter!
     Rword:= Rword + intToStr(ncount) + tmpStr[i] //+ last char(tmpStr)
end;
end; // end for loop
result:=Rword;
end;
```

UEB: 9_pas_umlrunner.txt

# After C.

```
function charCounter(instr: string): string;
var i, cntr: integer;
      Rword: string;
begin
cntr:= 1;
Rword:=' ';
  for i:= 1 to length(instr) do begin
    //last number in line
    if i= length(instr) then
      concatChars()
    else
    if (instr[i]=instr[i+1]) then cntr:= cntr +1
    else begin
      concatChars()
      //reinit counter!
      cntr:= 1;
    end;
  end; //for
 result:= Rword;
end;
```

UEB: 009_pas_umlrunner_solution_2step.txt

# Closure Advantage – We can avoid:

Bad Naming (no naming convention)

Duplicated Code (side effects)

Long Methods (to much code)

Temporary Fields (confusion)

Long Parameter List (Object is missing)

Data Classes (no methods)

- Large Class
- Class with too many delegating methods
- Coupled classes

UEB: 33_pas_cipher_file_1.txt

# Refactoring Closure

| Unit | Refactoring Function | Description |
|---|---|---|
| Package | Rename Package | Umbenennen eines Packages |
| Package | Move Package | Verschieben eines Packages |
| Class | Extract Superclass | Aus Methoden, Eigenschaften eine Oberklasse erzeugen und verwenden |
| Class | Introduce Parameter | Ersetzen eines Ausdrucks durch einen Methodenparameter |
| Class | Extract Method | Extract a Codepassage or define a Closure |
| Interface | Extract Interface | Aus Methoden ein Interface erzeugen |
| Interface | Use Interface | Erzeuge Referenzen auf Klasse mit Referenz auf implementierte Schnittstelle |
| Component | Replace Inheritance with Delegation | Ersetze vererbte Methoden durch Delegation in innere Klasse |
| Class | Encapsulate Fields | Getter- und Setter einbauen |
| Model | Safe Delete | Delete a class with reference |

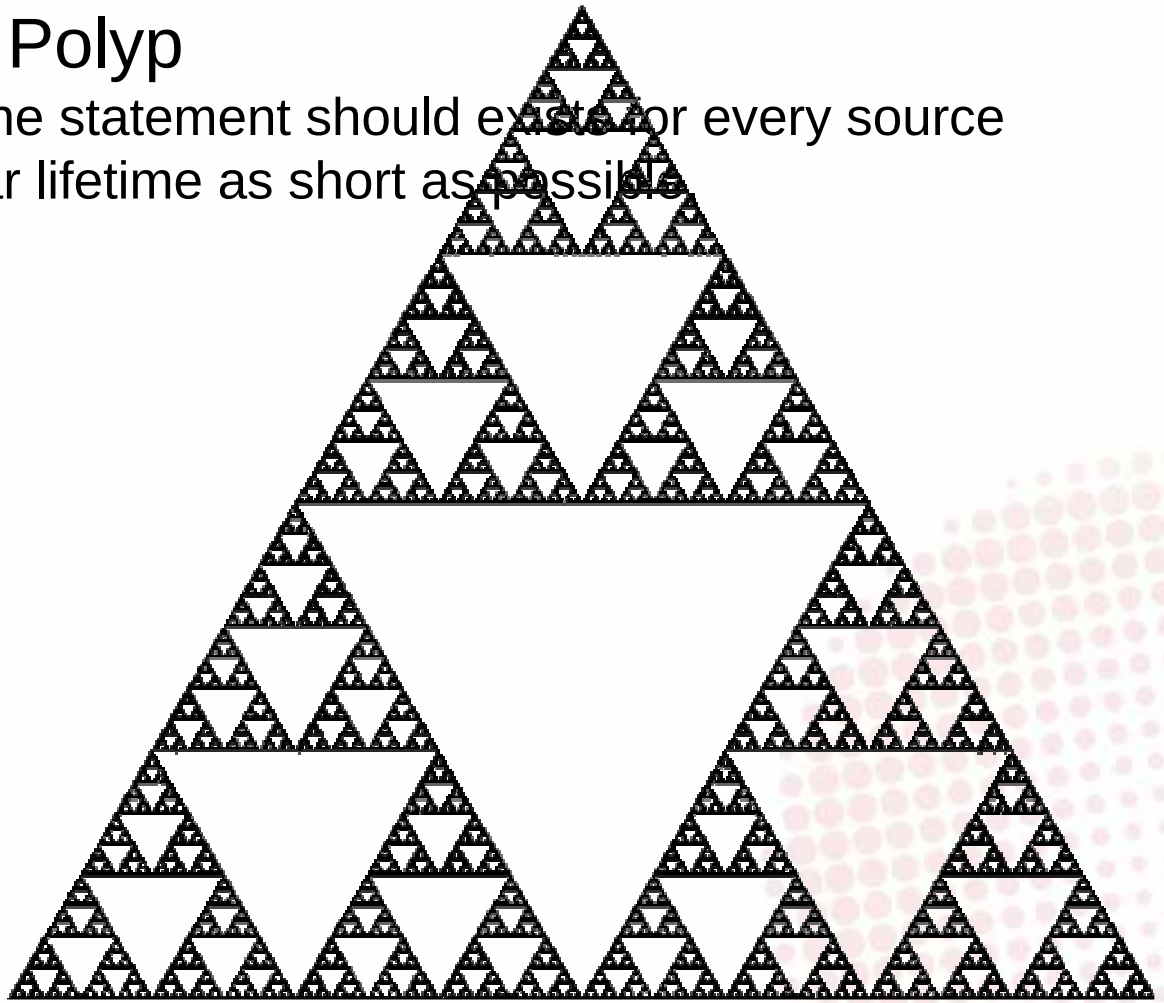
# Case Study: The Polygraph

- Design Case Study Polyp

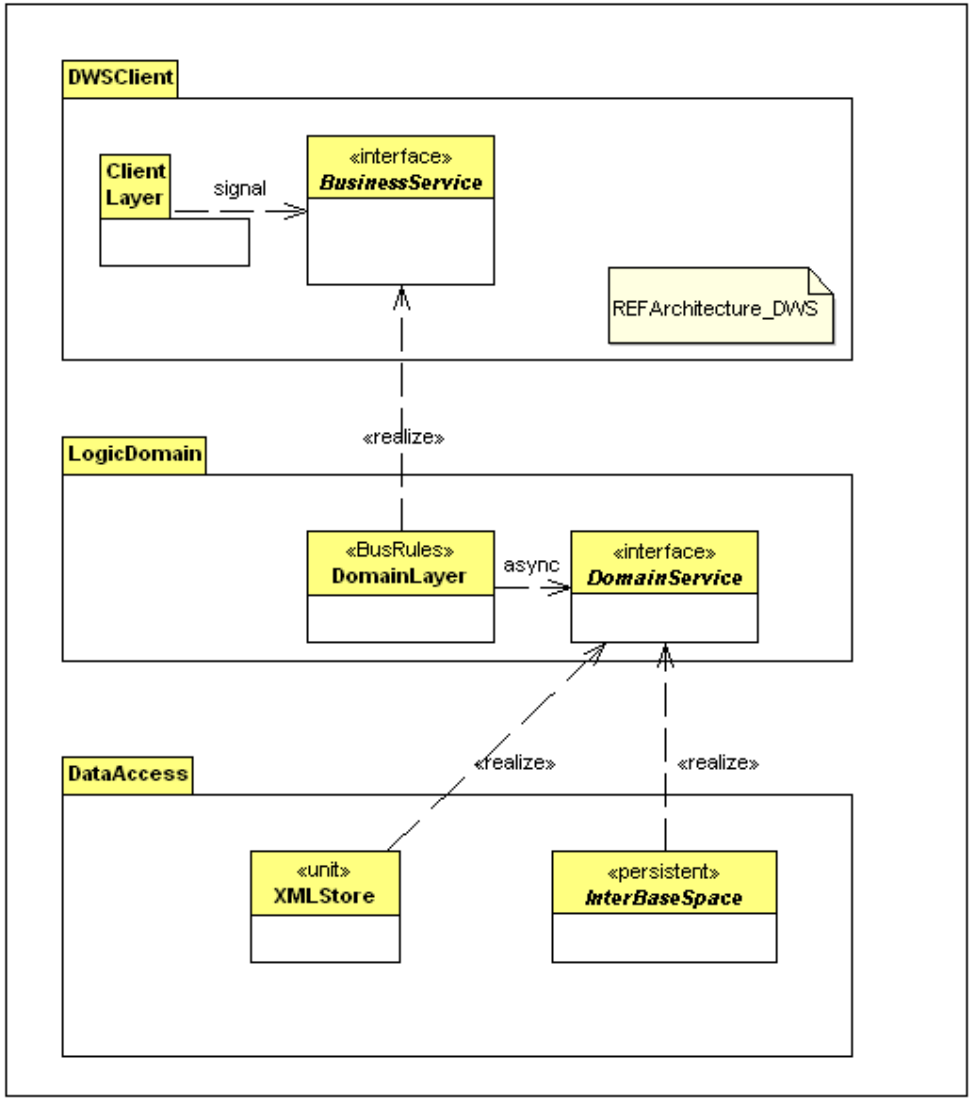
An easy one states that only one statement should exste for every source line, like next and put a Var lifetime as short as possible

Can you read (Yes-No)?
if YES then OK
if NO then you are a Liar!

```
var Rec: TMyRec;
begin ..
  with Rec do begin
  .. title:= ‘Hello Implikation';
  end;
end;
```



# Polyp Design




Dependency Inversion

# Final - **$Closure Test**

# Closure Links:

- http://gafter.blogspot.com/2007/01/definition-of-closures.html
- Michael Bolin, “Closure: The Definitive Guide", O'Reilly Media; Auflage: 1 (20. Oktober 2010).
- http://www.javac.info/
- http://sourceforge.net/projects/maxbox/
- *Refactoring* Martin Fowler (1999, Addison-Wesley)
- http://www.softwareschule.ch/download/closures_report.pdf
- http://www.refactoring.com

# *This is not The End:*
# *Q&A?*

max@kleiner.com
softwareschule.ch

maXbox