Patterns for Returning Multiple Results

ANDREW P. BLACK, Portland State University, USA

When a programming language does not support returning multiple results from a procedural abstraction, *i.e.*, a function, procedure, subroutine, or method, programmers must resort to patterns when they need to write a *multifunction*, that is, a function that naturally returns multiple results. Six such patterns are discussed, along with the context and forces that make each the solution of choice. Some of the patterns are obvious; some less so. Lambda-Expression Argument (Pattern 5) may be surprising to those who have not seen it before.

CCS Concepts: • **Software and its engineering** → **Software design tradeoffs**; **Software design engineering**; *Procedures, functions and subroutines*; Data types and structures; Classes and objects; **Patterns**.

Additional Key Words and Phrases: pattern, multiple results, multiple returns, once method, memo function

ACM Reference Format:
Andrew P. Black. 2024. Patterns for Returning Multiple Results. In *PLoP 2023: Pattern Languages of Programs Conference 2023.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Every programming language of practical use provides some kind of *procedural abstraction* — a way of encapsulating a formula or series of actions, and naming it for clarity and reusability. In object-oriented languages, such abstractions are known as methods; in procedural and functional languages they are called procedures, subroutines or functions, the latter in deference to mathematical functions.

Whatever they are named, *most* modern programming languages restrict these abstractions to returning a single result to the context from which they are "called". In this set of patterns, we will use the terms "call" and "called" consistently, even though a particular programming language might use another term, such as message send, method request, or invocation. Similarly, we will use the term function to refer to all kinds of procedural abstraction — unless there is a good reason to do otherwise. Each of these terms has its own connotations, and the devices that they denote do differ — but not in ways that are relevant to these patterns.

The problem that we address with these pattern is that of communicating the results of an n-way multifunction, that is, a function that naturally calculates n results, to its caller.

2 EXAMPLES OF THE PROBLEM

To make the problem, context and solutions concrete, we will use some simple running examples. The first example is the problem of dividing a file name, such as pattern.txt into two parts: a *base*, in this case "pattern", and an *extension*[1], in this case ".txt". Thus, there are two results, and computing one of them leads to the computation of the other. This is what we mean by a function "naturally" computing multiple results. For this example, the computation is simple; Listing 1 shows the code in the Grace programming language [Black et al., 2012].

[1]Some readers will have learned at their mother's knee that the *extension* part of a file name excludes the dot; if you are one of them, feel free to modify the example code accordingly. For our purposes, the exact definition matters not.

PLoP 2023, 22–25 October 2023, Monticello, Illinois, USA

Hillside ISBN 978-1-941652-19-0.
https://doi.org/10.1145/1122445.1122456

Listing 1: Sketch of Problem code in Grace

```
1 method baseAndExt(fileName:String) {       // return type omitted
2     def locationOfDot = fileName.lastIndexOf "." ifAbsent {fileName.size+1}
3     def base = fileName.substringFrom 1 to (locationOfDot−1)
4     def ext = fileName.substringFrom (locationOfDot)
5     return ...
6 }
```

Let's explain this example line-by-line. Because Grace is object-based, it uses the reserved word method[2] to introduce its procedural abstractions — the things that we will call *functions* in the context of this paper. On line 1 we define a method called baseAndExt(_) with a single parameter, whose type is declared to be String. (Type annotations are optional; they are signaled by a colon.) Line 2 defines the variable locationOfDot to be the result of a method on the fileName parameter. The name of the method is lastIndexOf(_)ifAbsent(_), which looks a little unusual, but is simply a method name, like size or baseAndExt(_); the parentheses (_) indicate the places where arguments are to be placed when the method is called. In this example, the first argument, ".", is the string for which we are searching, and the second, {fileName.size+1}, is the *code to be executed* if that string is not found.

Another way in which Grace differs from the Algol and C families of languages is that Grace does not always require parentheses around method arguments. In particular, the arguments "." and {fileName.size+1} are both self-delimiting, and wrapping them in additional parenthesis would add nothing but syntactic noise. In contrast, on line 3, the argument (locationOfDot−1) does need parenthesis. Grace strings are indexed from 1 to string.size, and substringFrom(_)to(_) does what its name implies, so line 3 binds the appropriate substring to the variable base. Line 4 is similar; the method substringFrom(_) (without a to(_) part) returns the substring that terminates at the end of the receiver.

After line 4, we have computed the two answers — that part was easy. What's not easy — and is the topic of this article — is deciding what to return. What is crucial about this example is that some part of the work is relatively expensive (here, locating the last occurrence of "." in fileName, which takes $O(n)$ time), and that once this is done, computing the multiple results is relatively cheap (here $O(1)$).

A second example comes from the div instruction in the Intel X86 instruction set, which divides one integer by another: it returns both the quotient and the remainder (in separate registers). The point here is that computing either quotient or remainder computes the other "for free". However, in almost[3] all high-level languages, the difficulty of returning multiple results means that there are separate functions for quotient and remainder, and the economy of computing both together is lost (or must be regained by a compiler optimization).

A third example is a function that computes the standard deviation of a set of numbers. In the process of so doing, it also computes the size of the set, its mean, and its variance.

This final example prompts an interesting question: are all of the results created equal, or are some more equal than others? This depends on one's expectations, which are largely determined by the name of the function. If I name my function standardDeviation, then I'm setting the expectation that its principal result will be the standard deviation of its argument (a collection of data): if other statistical measures are computed along the way, then they should either be dropped, or returned in some manner that makes them subsidiary. In contrast, if I name my function statisticalSummary, and have it compute the size of the dataset, its mean, and its variance as

[2]Object-centered programmers might be worrying: if baseAndExt(_) is a method, to what object is it attached? The answer is: the module in which it is situated. The whole Grace input file defines a singleton object (the module object) that provides a home for the methods and variables declared in the file.

[3]Lisp is an exception; the floor function returns both results.

well as its standard deviation, then I'm setting the expectation that all of these results will be treated equally. This article is primarily concerned with this latter case, in which none of the results is privileged above the others. However, "Out" Parameters (Pattern 4) and Lambda-Expression Argument (Pattern 5) can be used to return subsidiary results from a function whose return value communicates the primary result.

3 THE PATTERNS

All of these patterns address a similar problem: how do we return multiple results to the caller. In other words: what should we write in place of the ellipsis on line 5 of Listing 1?

Pattern 1: Native Multiple Returns

In the Introduction, we stated that most languages provide for the return of only a single result from a function. But *most* is not *all*: there are a few languages that do provide a native mechanism for returning multiple results. In the context of such a language, one should use the mechanism that it provides.

Context: *You are writing in a programming language that natively supports multiple return values.*

Problem: *You are writing a function that naturally returns multiple results, and do not wish to privilege one result over the others. How do you communicate the results of this function to its caller?*

Forces: *Part of the computation is necessary to produce more than one of the results, so computing one result produces the others as a by-product at small cost. The shared part of the computation is large; we wish to avoid repeating it.*

Solution: *Use the Native Multiple Return facility provided by your language.*

A language that does provide a mechanism for multiple results will typically also provide a way of binding or assigning these results to multiple variables. Amongst modern languages, Go and Dafny do support multiple results; here is our first example in Go.

Listing 2: Go code for `baseAndExt` with two results

```
1  func baseAndExt(s string) (string, string) {   // (string, string) says that this function returns two string results
2      locationOfDot := strings.LastIndex(s, ".")
3      base := s[0:locationOfDot]
4      ext := s[locationOfDot:]
5      return base, ext                           // return both values as the result
6  }
7  func main() {
8      b, e := baseAndExt("somefile.tex")         // access the return values using multiple assignment
9      ...
10 }
```

Why is this sort of facility not more common? A discussion of the issues will be found in Appendix A.

Consequences: *The consequences depend on the details of the Native Multiple Return facility provided by your language. In general, a good implementation of multiple returns ought to be more efficient than Structured Return Value (Pattern 2). This is because a structure will normally be allocated on the heap, whereas multiple function results can be implemented without creating a structure on the heap — for example, by returning the results in multiple registers, or on the stack.*

When working in a language that does *not* permit multiple results, we must resort to a programming pattern to represent a multifunction. The following five patterns can be used for this purpose.

Pattern 2: Structured Return Value

Context: *You are writing in a language that does not support multiple return values.*

Problem: *You are writing a function that naturally returns multiple results, and do not wish to privilege one result over the others. How do you communicate the results of this function to its caller?*

Forces: *Part of the computation is necessary to produce more than one of the results, so computing one result produces the others as a by-product at small cost. The shared part of the computation is large; we wish to avoid repeating it.*

Solution: *Program your function with a Structured Return Value that comprises the multiple results. For example, return a tuple, record, list, or trivial object, each of whose components or elements is one of the results. Return that structured value as the result of a normal, single-result function.*

The choice of exactly what structure to use depends on the programming language. In Pascal you might use a record; in C a struct; in Haskell or Python a tuple. In a language that permits heterogeneous[4] collections, we might use an array, list, or sequence, as does the example in Listing 3.

Listing 3: Grace code for `baseAndExt` returning a Sequence

```
1 method baseAndExt(fileName:String) {
2     def locationOfDot = fileName.lastIndexOf "." ifAbsent {fileName.size+1}
3     def base = fileName.substringFrom 1 to (locationOfDot−1)
4     def ext = fileName.substringFrom (locationOfDot)
5     return [base, ext]
6 }
```

The notation [base, ext] on line 5 constructs a sequence (that is, an immutable array) containing two elements. This is simple enough; what's not so simple is *using* the sequence. In many languages, including Grace, the components must be extracted from the sequence before they can be used. Listing 4 shows an executable specification (in the style of Behaviour-Driven Design) for Listing 3. To state our expectations of the two results, they must first be extracted from the sequence.

Listing 4: Grace client code using the definition from Listing 3

```
1     specify "base and extension of a dotted name" by {
2         def result = baseAndExt "pattern.txt"
3         expect (result.first) toBe "pattern"
4         expect (result.second) toBe ".txt"
5     }
```

The intermediate variable result is clumsy, and the accessors first and second are anything but mnemonic.

[4] In general, the *n* results may not all have the same type. Our running examples, in which both results are Strings, or all results are Numbers, may be atypical in this respect.

In a language that provides tuples, and allows them to be decomposed using a multiple assignment, things are better: one can use two mnemonic identifiers rather than one not-so-meaningful identifier. We could do this in Python: this small change (shown in Listing 5) yields a significant improvement in readability.

Listing 5: Python Code returning a Tuple

```
import pytest

def baseAndExt(fileName):
    locationOfDot = fileName.rfind(".")
    if (locationOfDot == -1):
        locationOfDot = len(fileName)+1
    base = fileName[0:locationOfDot]
    ext = fileName[locationOfDot:]
    return base, ext       # the comma creates a tuple

b, e = baseAndExt("pattern.txt")
assert b == "pattern"
assert e == ".txt"
```

This looks very much like the Go example from Listing 2. The difference is subtle: in Go there are two results, whereas in Python there is one (which happens to be a tuple). You might view this as solving the problem using an extra level of indirection: in Python we return one (structured) result, which indirectly gives us access to the two results we want, whereas in Go we actually return two results. What we can do with the result, or results, differs in consequence.

The use of an array or list to simulate a tuple is a code smell, especially when the results have different types. Note that when I use the term "code smell", I'm following Fowler [2006a]: a code smell is not necessarily a problem, just something that's a bit unpleasant, and that warrants a closer look.

In an object-oriented language, rather than (mis)-using an array, we might prefer to return a small custom object with public fields. In some languages, such as Smalltalk, this is clumsy, because it requires defining a whole class with its own factory method, as well as accessor methods for the fields, so in Smalltalk, using an array to return multiple results is an accepted idiom. Other languages make the task of defining an object easier. Java lets us write a slightly more lightweight *inner* class; JavaScript and Grace let us create and return a new object in an expression:

Listing 6: Grace code returning a custom object

```
method baseAndExt(fileName:String) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent {fileName.size+1}
    return object {
        def base is public = fileName.substringFrom 1 to (locationOfDot−1)
        def ext is public = fileName.substringFrom (locationOfDot)
    }
}
```

The reserved word object on line 3 introduces an object constructor, which manufactures a new object containing the attributes between the braces — here the public fields base and ext declared on lines 4 and 5.

Unfortunately, *using* a custom object is only slightly more convenient than using a tuple. We still need to name the result of the call, and then decompose it, as shown in Listing 7. An improvement is that the calls used to effect the decomposition now have meaningful names.

Listing 7: Grace client using the method of Listing 6

```
1  specify "base of a dotted name" by {
2      def result = baseAndExt "pattern.txt"
3      expect (result.base) toBe "pattern"
4      expect (result.ext) toBe ".txt"
5  }
```

Javascript is similar; just remember that the first character of a Javascript string is at index 0:

Listing 8: Javascript code returning a custom object

```
1 function baseAndExt(fileName) {
2   let locationOfDot = fileName.lastIndexOf(".");
3   if (locationOfDot == −1) locationOfDot = fileName.length + 1;
4   const b = fileName.substring(0, locationOfDot);
5   const e = fileName.substring(locationOfDot);
6   return {base: b, ext: e};
7 }
8 let {base: p, ext: t} = baseAndExt("pattern.txt");
9 console.info ("base = " + p + ", extension = " + t);
```

The let declaration on line 8, with what looks like a record constructor to the left of the =, is called a *destructuring assignment*; it extracts the base and ext fields from the result of the function call, and assigns them to the variables p and t, respectively. This obviates the need for the auxiliary variable result.

JavaScript has a special facility (usually called *shorthand properties*) intended to make this pattern briefer. When constructing an object, if the names of the fields are omitted, they are inferred from the name of the variables. Conversely, when *destructuring* an object, the names of the fields, if omitted, are assumed to be the same as the names of the variables. Hence, the following code is equivalent to the above:

Listing 9: Alternative JavaScript with the same effect as Listing 8

```
function baseAndExt(fileName) {
  let locationOfDot = fileName.lastIndexOf(".");
  if (locationOfDot == −1) locationOfDot = fileName.length + 1 ;
  const base = fileName.substring(0, locationOfDot);
  const ext = fileName.substring(locationOfDot);
  return {base, ext};
}

let {base, ext} = baseAndExt("pattern.txt");
console.warn ("base = " + base + ", extension = " + ext);
```

In spite of the brevity, I dislike this facility, because it is a pun on the names of the variables. A refactoring that changes the name of a local variable uniformly within its scope will break this code, because the names of the object's properties will no longer match the names of the local variables. Such code gains some brevity for the writer, at the cost of readability and maintenance headaches. This is not a good trade-off.

Consequences: *If the structured return value is a tuple, the results do not have names. If the return value is a record, a custom object, or a dictionary, then the results must be given names by the function designer: these names become the record selectors of the record, the messages understood by the object, or the keys to the dictionary. These names must be made known to all clients. If you use a destructuring assignment with shorthand property names, your code may become hard to read for the uninitiated, and brittle to seemingly-harmless rename-variable refactorings.*

Pattern 3: Domain Object

The final variant of Structured Return pattern uses a custom object to encapsulate the data that the multiple-result-function needs to return. However, we know that objects should be used to encapsulate not just data, but also computations — in this case, the computations that create and manipulate that data. We can apply that knowledge and move the multi-function into the object, where it becomes *multiple* methods. For our running example, the code is only slightly different from the previous solution, but the organization and the intent are dramatically different.

Context: *You are programming in an object-oriented language.*

Problem: *You are writing a function that naturally returns multiple results, and do not wish to privilege one result over the others. How do you communicate the results of this function to its caller?*

Forces: *Several computations will be performed on the same data, and similar data occurs repeatedly in the problem domain. The shared part of the computation is large; we wish to avoid repeating it.*

Solution: *Create a Domain Object and represent each computation as a method thereon. Put the argument to the original multifunction into an object that represents an entity in the problem domain. Provide n methods on that object, each of which returns one of the n results, along with methods and fields for other computations, including the shared part of the original computation.*

Here is our running example coded using this pattern. The fileName argument becomes an initialization argument to the class that constructs the object — it is the datum that the object encapsulates. Extracting the base component and the extension component from that name are two computations that can be performed on it. This pattern shows its value when there are many operations, such as checking to see if the file exists in the current directory,

searching for it along a path, interrogating file metadata (size, time last written, *etc.*) and obtaining the contents of the file; the reader is invited to imagine adding such operations to Listing 10.

Listing 10: Defining a Class of Domain Objects in Grace

```
1 class fileName(name:String) {
2     def locationOfDot = name.lastIndexOf "." ifAbsent {name.size+1}
3     method base { name.substringFrom 1 to (locationOfDot−1) }
4     def ext is public = name.substringFrom (locationOfDot)
5     // other methods on fileNames
6 }
```

Notice on line 1 that what was previously a *method* is now a *class*, and is now called fileName, to convey the intention that objects of this class represent file names. (In Grace, a class is a method that returns a new object.) That class is parameterized by the initial datum — the file's name — so that this datum is accessible to the code of the object created by the class.

The computation of the components of the name, base and exp, executes the same code as before, but the code is organized differently. There are many ways of organizing the code inside an object; our example illustrates some particular choices, but other choices are available, and may be preferable, in certain circumstances.

On line 2, the initialization code of the object being created by the class computes, once and for all, the location of the dot. Thus, the shared part of the computation is executed just once. Clients of this object can use base and ext to access the components:

Listing 11: Grace client using an object defined using Listing 10

```
1     specify "base of a dotted name" by {
2         def fn = fileName "pattern.txt"
3         expect (fn.base) toBe "pattern"
4         expect (fn.ext) toBe ".txt"
5     }
```

To illustrate the possibilities, on line 3 of Listing 10 we declare base as a method (which in Grace is public by default), whereas on line 4 we declare ext as a public field. You might ask: which is the better choice?

Recall that the motivation for all of the patterns that we have seen so far is to avoid unnecessarily repeating some computation — in the filename-splitting example, the searching of the filename for the dot. One of the advantages of the Domain Object pattern is that it gives the programmer fine control over when and if the various parts of the computation are executed. If the attribute is a method, as with base on line 3 of Listing 10, the substring computation in the body of that method will be performed once each time a client accesses base. In contrast, If the attribute is a public variable, as with ext on line 4, the substring computation is performed as part of the process of initializing the object. Thus, *its* substring computation is executed exactly once, whether a client accesses ext zero, three or fifty times. If the individual computations are small and constant time, as here, the choice hardly matters — provided that your language supports what Doug Ross [1970] dubbed, at the first Software Engineering Conference, "The Uniform Referent Property". This property means, amongst other things, that whether an attribute of an object is computed (as is base), or stored (as is ext), makes no difference to the way that a client *refers to* that attribute. (Bertrand Meyer [1997, p.57], called what seems to be the same idea the

"Uniform Access Principle".) As a consequence, the implementor of an object is free to change a public variable into a method, or vice versa, without affecting the code of its clients.

However, if the individual computations are expensive, neither option is ideal: using a method may repeat the computation, while using a variable may perform it unnecessarily. The best choice might then be to make the accessors *memo functions* [Michie, 1968], which perform their computations the first time they are called, but *memoize* their results in variables so that, if they are called again, the previously computed result can be returned immediately. Memo functions can be implemented in any language using the following pattern, which we exemplify in JavaScript[5] in Listing 12.

Listing 12: Implementing a memo function in JavaScript using a *getter*

```
class Filename(fn)
    constructor {
        this.fn = fn;
    }
    get ext() {                    // a "getter" – a method that is used like a field
        if (this.extCache == undefined) {   // idiomatic JavaScript would say "! this.extCache"
            // perform computation of ext and assign to extCache
        }
        return this.extCache;
    }
```

Lazy initialization of instance variables in Smalltalk [Beck, 1997, p. 85] uses the same idea, and is implemented by the programmer with similar code. A few languages, notably Eiffel [Meyer, 1992, p.113] and Grace, support "once" routines or methods, which tell the compiler to implement the cache for you. Listing 13 shows the fileName class in Grace using once methods.

Listing 13: Grace Class using Once Methods

```
class fileName(name:String) {
    def locationOfDot = name.lastIndexOf "." ifAbsent { name.size+1 }
    once method base { name.substringFrom 1 to (locationOfDot−1) }
    once method ext { name.substringFrom (locationOfDot) }
}
```

Once methods are so easy to use that we might decide to make locationOfDot a once method too; this is a reasonable decision if the fileName object has many operations, and not just the two that we have been using in our example. Why? Because the fileName object might have been created so that these *other* operations can be called, and base and ext may never be called, in which case locationOfDot need not (and will not) be evaluated.

Consequences: *The cost of this pattern is that the client code must be reorganized to first create the domain object, and then access its attributes. The benefit is that the programmer has precise control over when the various pieces of the computation are executed. By using memoization (*once *methods), the programmer can*

[5]It's not quite so simple in JavaScript, which does *not* enjoy the Uniform Referent Property: JavaScript uses different syntax for accessing a field of an object and calling a parameterless method on the object. We circumvent this problem by defining ext as a Javascript *getter*, which is has a body like a method, but is invoked with the same syntax as is used to access a field.

eliminate unnecessary and repeated computation. If the programming language does not support Uniform Referents, then turning fields into methods may require changes in the client.

Pattern 4: "Out" Parameters

Some programming languages, such as Algol 60 (with call-by-name), Fortran (with call-by-reference), and Ada (with **out** parameters), allow a function to modify a variable passed as an argument. In such languages, baseAndExt would have two additional parameters, with the expectation that the corresponding arguments will be variables, and that execution of the function body will bind them to new values. Most modern languages don't support out parameters, but the same effect can be obtained by passing a one-element array, or a *value holder*[6]. The latter is an object that does little but provide storage for, and access to, another object: the *value*.

Context: *You are writing a in a programming language that provided out parameters, or a language in which idioms that simulate out parameters are common.*

Problem: *You are writing a function that naturally returns multiple results, and do not wish to privilege one result over the others. Or, you are writing a function that returns a single privileged result, but also wish to communicate additional results. How do you communicate the results of such a function to its callers?*

Forces: *Several computations will be performed on the same data. The shared part of the computation is large; we wish to avoid repeating it. The results of the multifunction will be used several times, so they need to be stored in variables.*

Solution: *Use "out" parameters. Add n additional parameters to the n-way-multifunction, to be used for transmitting the results. Make them* output *parameters, or a simulation of output parameters. The client code declares n additional variables to hold the results.*

Listing 14 an implementation of baseAndExt that expects as additional arguments two value holder objects to capture the results.

Listing 14: Grace simulation of out parameters using ValueHolders

```
type ValueHolder⟦T⟧ = Object & interface {
    value → T                // return my value
    value:=(new:T)           // change my value
}

method baseAndExt(fileName:String) output (outBase:ValueHolder⟦String⟧, outExt:ValueHolder⟦String⟧) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent { fileName.size+1 }
    outBase.value := fileName.substringFrom 1 to (locationOfDot−1)
    outExt.value := fileName.substringFrom (locationOfDot)
}
```

The client code (Listing 15) must declare and create the actual value holders before calling the multifunction. Notice also that the call of the multifunction no longer returns an interesting value, so it is used as a statement, not as an expression. In other words, it is no longer a "function" in the mathematical sense, but a "procedure", called for its effect.

[6]VisualWorks Smalltalk provided a ValueHolder class; this is mentioned by Fowler [2006b] and Alpert et al. [1998], although neither reference describes it in any detail. The VisualWorks Cookbook [1995] provides some recipes for ValueHolders in presentation models.

Listing 15: Grace client code using the definitions of Listing 14

```
1  class valueHolder⟦T⟧ → ValueHolder⟦T⟧ {
2      var value is public
3      method asString { "value [{value}]" }
4  }
5  describe "filename base and extension" with {
6      specify "base of a dotted name" by {
7          def baseHolder = valueHolder⟦String⟧
8          def extHolder = valueHolder⟦String⟧
9          baseAndExt "pattern.tex" output (baseHolder, extHolder)
10         expect (baseHolder.value) toBe "pattern"
11         expect (extHolder.value) toBe ".tex"
12     }
13 }
```

However, out parameters have fallen out of favour for a reason: they are hard to read. The fact that an argument is being used to move information *out* of a function, rather than *in* to it, is far from apparent from the function call (although the Grace example in Listing 14 gives us a hint by using the word "output" in the name of the method). Out parameters are also non-compositional: this means that the call containing the out parameters cannot be used as an expression (or, more precisely, if it is so used, the result is not available). Both the lack of readability and the non-compositionality stand in contrast to function results, where using the call in an expression context clearly communicates that its value is being consumed by some other program element.

Consequences: *The caller of the n-way-multifunction has the additional burden of declaring n output variables. The multifunction is no longer an expression that returns a value, but a statement that has the effect of changing the output variables. Compositionality has been lost; the code is less readable, and more verbose.*

The problem statement for this pattern mentions that it can also be applied in a situation where one result is primary, but some subsidiary results are also made available. This is illustrated in Listing 16, where a method calculates and returns the naïve standard deviation of a collection of numbers, but also passes back their sum, mean and variance using (simulated) out parameters. The sum, mean and variance are computed on lines 6–9, the first four lines of the method standardDeviation(_)sum(_)mean(_)variance(_). Line 10 invokes the set methods (denoted by .value :=) of the three valueHolder objects. Then, on line 11, we return the standard deviation as the square-root of the variance.

Listing 16: Grace standard deviation method; VaueHolders pass three subsidiary results

```
1  // definitions of the ValueHolder type and the valueHolder class from Listings 14 and 15
2  method standardDeviation (data:Collection⟦Number⟧)
3          sum (sum:ValueHolder⟦Number⟧)
4          mean (mean:ValueHolder⟦Number⟧)
5          variance (v:ValueHolder⟦Number⟧) → Number {
6    def sum = data.fold {acc, item → acc + item} startingWith 0
7    def n = data.size
8    def mean = sum / n
9    def variance = data.fold {acc, item → acc + ((item − mean) ^ 2)} startingWith 0 / n
10   s.value := sum ; m.value := mean ; v.value := variance
11   return variance.sqrt
12 }
13 describe "standardDeviation" with {    // this executable specification shows how the method is used:
14   def s = valueHolder⟦Number⟧
15   def m = valueHolder⟦Number⟧
16   def v = valueHolder⟦Number⟧
17
18   specify "statistics of 2 4 4 4 5 5 7 9" by {
19     def sd = standardDeviation [2, 4, 4, 4, 5, 5, 7, 9] sum (s) mean (m) variance (v)
20     expect (m.value) toBe 5
21     expect (v.value) toBe 4
22     expect (sd) toBe 2
23   }
24 }
```

Pattern 5: Lambda-Expression Argument

Most modern programming languages provide a way to create anonymous functions "on the fly" by enclosing the code that computes the function in some special syntax. The notation for creating such anonymous functions is often referred to as a λ-expression, or just a Lambda, after the notation that serves this purpose in Church's λ-calculus. Smalltalk, Java, JavaScript, Python, C++, and C# all support λ-expressions, although the syntax differs. For example, for parameter x, Java uses (Type x) -> body, Javascript uses (x) => body, Python uses lambda x: body, and Grace uses {x → body}. Here is a Grace λ-expression for the function that sums its arguments:

```
{ n, m → m + n }
```

The parameter list n, m is optional; if there are parameters, they are separated from the body of the function by an arrow. You already saw the use of a parameterless λ-expression as an argument on line 2 of Listing 1 (on page 2). There, the second (ifAbsent) argument to fileName.lastIndexOf(_)ifAbsent(_) is {fileName.size+1} — a parameterless λ-expression that provides a result for the function when no dot is found in fileName.

Although designed to pass functions *into* other functions, it so happens that λ-expressions also provide a convenient way of getting multiple results *out* of a function.

Context: *You are writing in a programming language with "lambdas": a concise notation for writing anonymous functions in-line.*

Listing 17: The λ-expression Argument Pattern in Grace

```
1  method baseAndExt(fileName:String) in (continuation:Function2) {
2      def locationOfDot = fileName.lastIndexOf "." ifAbsent {fileName.size+1}
3      def base = fileName.substringFrom 1 to (locationOfDot−1)
4      def ext = fileName.substringFrom (locationOfDot)
5      return continuation.apply(base, ext)
6  }
7  describe "file name properties" by {
8      specify "base of a dotted fileName" by {
9          baseAndExt "pattern.tex" in { b, e →
10             expect (b) toBe "pattern"
11             expect (e) toBe ".tex"
12         }
13     }
14 ...
15 }
```

Problem: *You are writing a function that naturally returns multiple results, and do not wish to privilege one result over the others. Or, you are writing a function that returns a single privileged result, but also wish to communicate additional results. How do you communicate the results of such functions to their callers?*

Forces: *Several computations will be performed on the same data. Choosing not to return multiple results would cause unnecessary repeated computation. Creating an object to encapsulate multiple results seems inappropriate, because there is no abstraction for the object to capture.*

Rather than trying to explain this solution in the abstract, let's look at the filename-splitting example using this pattern. Listing 17 shows the definition of the function, now called baseAndExt(_)in(_), and a use of the function in an executable specification. Notice that the definition of baseAndExt(_)in(_) is very similar to that of baseAndExt(_) in our first example (Listing 1 on page 2). There are two differences.

(1) The method has an additional parameter, named continuation, which is a function of two arguments.
(2) In place of the ellipsis on line 5, we *apply* that parameter to the results base and ext.

To see why this works, look at the call of baseAndExt(_)in(_) on lines 9–12 in the specify statement. Notice that the uses of the two results b and e — here simple checks that their values are as expected — are in the *body* of the λ-expression that we pass as the continuation argument.

When this λ-expression is applied, the parameters b and e are bound to the arguments base and ext *inside* the method baseAndExt(_)in(_), and then the expect ... statements are executed. Finally, the method baseAndExt(_)in(_) itself returns with the value returned by the execution of continuation. (Whether this is useful is up to the client; in this example, it is not, because expect(_)toBe(_) does not *return* an interesting result.)

Now you may begin to see why we called the extra function parameter continuation, and introduced it with the in part of the function name: it encapsulates code that *continues* the execution of the function. Indeed, the whole computation that is to follow the function can appear *in* the body of the λ-expression, in this case all of the program that follows the call to baseAndExt moves one level of indentation to the right, but we can still think

of it as simply following the call to baseAndExt. This is similar to an *if-then-else-if* ..., where the second if is syntactically nested, but we can still think of the whole as a multi-way branch.

However, this is not the only way in which the caller can use the λ-expression argument. The caller might simply assign the results to local variables, and then use those variables at the outer level, like this:

Listing 18: Alternative use of λ-expression argument in Grace

```
1  ...
2      specify "base of a dotted fileName" by {
3          var base
4          var ext
5          baseAndExt "pattern.tex" in { b, e →
6              base := b
7              ext := e
8          }
9          expect (base) toBe "pattern"
10         expect (ext) toBe ".tex"
11     }
12 ...
```

Note that some languages (notably Java) place restrictions on their λ-expressions that prohibit updating variables from the enclosing scope. As a consequence, the Java equivalent of Listing 18 is illegal.

Now we can describe the solution.

Solution: *In the definition of the n-way-multifunction, add an additional parameter called* continuation*, which will be a function of n arguments. Inside the body of the multifunction, after the computation is complete, apply* continuation *to the n results. To use the multifunction, the client passes an additional* Lambda-Expression Argument *as the continuation parameter; the n parameters of the λ-expression are fresh variables that will be bound to the n results.*

Consequences: *We have solved the problem of returning n results: they are bound to the variables in a new scope that is lexically nested in the calling scope. The programmer of the client can choose names for the parameters that will be bound to the results, and can also annotate them with types if they so wish (and the language permits). The client operates on these parameters within the scope of the λ-expression. The client may choose to simply bind the values of the parameters to names declared in an enclosing scope (if the language so permits), or the client may fully embrace* continuation-passing style*, and place the rest of the computation inside the body of the λ-expression.*

As with "Out" Parameters (Pattern 4), this pattern can also be used when there is a single primary result and one or more ancillary results. Listing 19 shows the standard deviation method from Listing 16 written using a λ-expression parameter. As before, the sum, mean and variance are computed on the first four lines of the method standardDeviation(_)in(_) (lines 2–5). Line 6 applies the continuation to the arguments sum, mean, and variance, and then on line 7, the standard deviation is computed as the square-root of the variance, and returned.

Listing 19: Grace standard deviation method; λ-expression captures subsidiary results

```
1  method standardDeviation (data:Collection⟦Number⟧) in (continuation:Procedure3) → Number {
2    def sum = data.fold {acc, item → acc + item} startingWith 0
3    def n = data.size
4    def mean = sum / n
5    def variance = data.fold {acc, item → acc + ((item − mean) ^ 2)} startingWith 0 / n
6    continuation.apply(sum, mean, variance)
7    return variance.sqrt
8  }
9
10 describe "standardDeviation" with {      // this executable specification shows how the method is used:
11   specify "statistics of 2 4 4 4 5 5 7 9" by {
12     def sd = standardDeviation [2, 4, 4, 4, 5, 5, 7, 9] in { _, m, v →
13       expect (m) toBe 5
14       expect (v) toBe 4 }
15     expect (sd) toBe 2
16   }
17 }
```

Notice that when standardDeviation is called (Lines 12–15), the λ-expression argument need bind only those results that are actually needed (here, the mean and the variance).

Pattern 6: Single-Result Function

The computation in our multifunction can be though of as comprising two parts.

(1) The shared part, which contributes to all of the results. In our file name example, this is the searching of fileName for the dot on line 2 of Listing 1.
(2) The individual parts, which contribute the calculation of just one of the individual results. In our running example, these are the selection of the substrings on lines 3 and 4.

So far, we have assumed that the case for a multifunction is compelling: the shared work is too costly to repeat. If a Domain Object is waiting to be discovered, then Domain Object (Pattern 3) suggests putting the *shared part* into a memo function, and making each of the *individual parts* a separate accessor function.

However, before adopting any of the other patterns, we should ask ourselves whether the gain in efficiency is really worth the pattern's cost in complexity. Suppose we just repeat the shared part? Is the amount of shared computation in the multifunction so great that we can't afford to do it twice? In our toy example, the answer is clearly no. We can easily replace the single multifunction baseAndExt by two separate functions, base, and ext, as shown in Listing 20.

Listing 20: Two single-result methods in Grace

```
method base(fileName:String) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent { fileName.size+1 }
    def base = fileName.substringFrom 1 to (locationOfDot−1)
    return base
}

method ext(fileName:String) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent { fileName.size+1 }
    def ext = fileName.substringFrom (locationOfDot)
    return ext
}
```

This is our final working example. The complete program, with an executable specification, is shown in Listing 21. If you wish, you can run this in a web browser by visiting http://www.cs.pdx.edu/~grace.

Context: *The shared part of the computation is small, either in absolute terms, or relative to the individual parts. Some uses of the multifunction require only a subset of the results.*

Problem: *You are writing a function that seems to require the return of multiple results, and do not wish to privilege one result over the others. How do you communicate the results of this function to its caller?*

Forces: *Several computations will be performed on the same data. The shared part of the computation is small; clarity of expression and reduced maintenance costs are more important than efficiency of execution.*

In this context, the solution is so obvious that it is easy to overlook.

Solution: *Instead of writing a single multifunction that calculates all the results, write one* Single-Result Function *for each result.*

Consequences: *The client must be aware of two simple functions, rather than one more complex multifunction. The shared part of the computation is repeated, with some cost in efficiency. The implementor may need to perform an extract method refactoring to eliminate the duplicated code introduced by repeating the shared part of the computation.*

Listing 21: Complete Grace program using the Single Result Function Pattern

```
dialect "minispec"

method base(fileName:String) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent { fileName.size+1 }
    def base = fileName.substringFrom 1 to (locationOfDot−1)
    return base
}

method ext(fileName:String) {
    def locationOfDot = fileName.lastIndexOf "." ifAbsent { fileName.size+1 }
    def ext = fileName.substringFrom (locationOfDot)
    return ext
}

describe "filename base and extension" with {
    specify "base of a dotted name" by {
        expect (base "pattern.txt") toBe "pattern"
    }

    specify "extension of a dotted name" by {
        expect (ext "pattern.txt") toBe ".txt"
    }

    specify "base of a non−dotted name" by {
        expect (base "/usr/local/pattern") toBe "/usr/local/pattern"
    }

    specify "extension of a non−dotted name" by {
        expect (ext "/usr/local/pattern") toBe ""
    }

    specify "extension of an empty extension" by {
        expect (ext "/usr/local/pattern.") toBe "."
    }
}
```

4 SUMMARY

We have introduced the idea of a *multifunction* as a function that naturally needs to return multiple results. We have presented patterns for six possible solutions to the problem of returning results from a multifunction.

(1) Native Multiple Returns
(2) Structured Return Value
(3) Domain Object
(4) "Out" Parameters
(5) Lambda-Expression Argument
(6) Single-Result Function

Of these patterns, Single-Result Function (Pattern 6) should be considered first. If you decide that you really need to combine the computations, then consider creating a Domain Object (Pattern 3). If there is no domain abstraction to capture, I advise that you return the results with a Lambda-Expression Argument (Pattern 5).

The facilities available in your programming language will also influence the pattern that you choose. In the case where the language supports multiple results directly, use Native Multiple Returns (Pattern 1). If the language does not support λ-expressions, Lambda-Expression Argument (Pattern 5) will not be possible. Special object creation and destructing shorthands, such as JavaScript's shorthand property names (see page 6), may influence you to use Structured Return Value (Pattern 2). If your language provides them, you may choose to use "Out" Parameters (Pattern 4).

The reader may note that Global Variable is *not* one of these patterns. Indeed, it is an anti-pattern: please don't go there.

ACKNOWLEDGMENTS

To Rebecca Wirfs-Brock, for convincing me to write down these patterns, and to Allen Wirfs-Brock, for useful advice that has helped improve the presentation. To James Noble, who suggested citations and some restructuring, and to Richard Gabriel, my PLoP Shepherd, who showed a personal interest in these patterns and helped improve them in many ways. I am also indebted to the members of the *Seljuk Being* writers workshop at PLoP, who made many invaluable suggestions: Indu Alagarsamy, Marden Neubert, Douglas C. Schmidt, Mary Shaw, Kurihara Wataru, and Joe Yoder.

REFERENCES

Sherman R. Alpert, Kyle Brown, and Bobby Woolf. 1998. *The Design Patterns Smalltalk Companion.* Addison-Wesley Longman Publishing Co., Inc., USA.

Kent Beck. 1997. *Smalltalk Best Practice Patterns.* Prentice Hall, Upper Saddle River, NJ. ISBN 0-13-476904-X.

Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software.* ACM, New York, NY, 85–98. http://doi.acm.org/10.1145/2384592.2384601

Martin Fowler. 2006a. CodeSmell. https://martinfowler.com/bliki/CodeSmell.html https://martinfowler.com/bliki/CodeSmell.html.

Martin Fowler. 2006b. GUI Architectures. https://martinfowler.com/eaaDev/uiArchs.html#VisualworksApplicationModel

Bertrand Meyer. 1992. *Eiffel: The Language.* Prentice Hall.

Bertrand Meyer. 1997. *Object-Oriented Software Construction* (second ed.). Prentice Hall Professional Technical Reference.

Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218 (6 April 1968), 19–22. https://doi.org/10.1038/218019a0

R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. 1991. Emerald: A General Purpose Programming Language. *Software—Practice and Experience* 21, 1 (1991), 91–118.

Douglas T. Ross. 1970. Uniform Referents: An Essential Property for a Software Engineering Language. In *Software Engineering*, Julius T. Tou (Ed.). Vol. 1. Academic Press, 91–101.

Guy L. Steele Jr. and Richard P. Gabriel. 1996. The Evolution of Lisp. In *History of Programming Languages II*, Thomas J. Bergin and Richard G. Gibson (Eds.). Chapter VI, 233–330.

VisualWorks. 1995. *Visualworks Cookbook* (rev 2.0 ed.). ParcPlace-Digitalk, Chapter 32: Setting up Simple Value Models (ValueHolder), 704–705. http://www.esug.org/data/Old/vw-tutorials/vw25/cb25.pdf

Appendix A HOW DID WE GET HERE?

The motivation for these patterns is that most modern languages do not natively support returning multiple results. Why is this so? If patterns are devised to overcome shortcomings in programming languages, might we expect future languages to support multiple results? This has happened with the iterator pattern, with the singleton pattern, and with memo functions: all have become language features in certain recent languages.

In addition to Go and Dafny, already mentioned, Common Lisp supports multiple results. Thus, in Common Lisp one might write:

Listing 22: Multiple value return and assignment in Lisp

```
(defun baseAndExt (filename)
   (let ((locationOfDot (position #\. filename :from-end t)))
      (cond ((numberp locationOfDot)
               (values
                  (subseq filename 0 locationOfDot)
                  (subseq filename locationOfDot)))
            (t (values (subseq filename 0 locationOfDot) "")))))
(let (b e)
   (multiple-value-setq (b e) (baseAndExt "Pattern.txt"))
   (format t "base = ") (format t b) (format t "~%")
   (format t "extension = ") (format t e) (format t "~%")
)
```

As in Go, the two results are returned directly, without constructing a list or other form of indirection. But syntactically, multiple results are not "first class": one has to use the special form (values ...) to "construct" the multiple results, and (multiple-value-setq ...) to assign them. Why not take multiple results a step further, and allow every function to return multiple results without special syntax, and also allow multiple values to be bound to a *single* variable?

Several language designers have independently pursued this path, and quietly abandoned it. The following email exchange between Andy Freeman and Guy Steele from 1986 documents one such episode:

Guy Steele on MARVEL

> Date: Sat 1 Nov 86 09:14:39-PST
> From: Andy Freeman <ANDY@Sushi.Stanford.EDU>
>
> Apparently you wrote a language that had a multiple values scheme much like the one I attributed to Talcott and Weyrauch. Can you comment on it to rrrs-authors? (T&W like it and have stayed with it; you abandoned it. I'm sure both had their reasons.)

Yes. For a while I worked on a dialect of Scheme called MARVEL (Multiple-Return-Value-Expression Lisp; if you ask "where does the 'A' come from?" I say "A is for Acronym"). It did pretty much all the obvious things: every function call was implicitly like the Common Lisp MULTIPLE-VALUE-CALL, and most side-effecting forms such as SETQ, PRINT, and COMMENT were made to return zero values. I believe I also arranged for variables to be able to hold multiple values. My experience with the language was that it was perfectly clean and elegant, but programs that made non-trivial use of multiple values were very hard to read, precisely because of the loss of the one-form/one-value correspondence. Having the extra power everywhere in the language was not worth the loss of clarity. I therefore abandoned the experiment without writing it up.

Steele and Gabriel discuss the design of Multiple Value Lisps in their HOPL II paper "The Evolution of Lisp" [1996, §6.3.5.2]. They comment:

> … nearly everyone who has followed this path has given up at this point in the development, muttering, "This way madness lies," and returned home rather than fall into the tarpit. …
>
> Among the most notable efforts that did produce actual implementations before eventual abandonment are SEUS and POP-2 [Burstall 1971]. The designers and implementors of SEUS (Len Bosack, Ralph Goren, David Posner, William Scherlis, Carolyn Talcott, Richard Weyhrauch, and Gabriel) never published their results, although it was a novel language design, and had a fast, compiler- and microcode-based implementation, complete with programming environment.

The Emerald Programming Language [Raj et al., 1991] also allowed multiple results from methods, and multiple assignment, but did not generalize to allow multiple values to be bound to a single variable. This meant that a method returning multiple results could stand on the right-hand-side of an assignment, but could not be used in other expression contexts. It thus avoided Steele and Gabriel's tarpit, but also failed to capitalize on the potential power of this generalization. Whether a worthwhile generalization will ever be found remains an open question.