JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2009

Vol. 8, No. 6, September–October 2009

Functional Programming at Work in Object-Oriented Programming

Ph. Narbel, LaBRI, University of Bordeaux 1, France

This paper is a synthesis about why and how some functional programming (FP) can be helpful from a program design point-of-view within mainstream object-oriented programming (OOP). We first introduce criteria to ensure that FP-oriented features give an effective functional/method granularity design level within OOP. Next, we list up and discuss the general techniques and design consequences of having such FP capabilities in OOP, i.e. code abstraction/factoring at a function level, generic iterator/loop implementations, operation compositions, sequence comprehensions, partial application and currying, reduction of the number of class definitions, name abstractions, and function-based structural compatibilities. We also stress some of the difficulties of blending FP constructs with OOP, by pointing out potential problems related to *design granularity mismatch*, architecture non-uniformity and datatype incoherences. Several classic OOP design patterns are analyzed, since FP techniques make alternative implementations manageable: basic cases like Strategies, Commands and Observers, but also Proxies (using functional-based evaluation control) and Visitors (using functional data-driven programming). This synthesis is illustrated with C# 3.0 which offers effective FP-oriented features – based on *delegates* –, but also by using comparisons with other cross-paradigm languages.

1 INTRODUCTION

Let us consider the following claim:

Adding some functional programming capabilities within an object-oriented language leads to benefits for object-oriented programming design.

The purpose of the following pages is to provide a synthetic and practical presentation of the pros and cons about this claim. First, with regard to functional programming (FP) mixed with object-oriented programming (OOP), several points are worth recalling to get a picture of the situation: *(1) Existence of FP+OOP languages*: Successful languages allowing programmers to mix FP and OOP have been available, some for a long time like Smalltalk and Common Lisp (CLOS), and some more recently like Python and Ruby. Also, some modern languages with sophisticated typing and modular systems are being developed offering very effective mixes of FP and OOP (see e.g. OCaml [LDG+08] and Scala [Ode08]). *(2) Existence of FP-OOP comparisons points*: in the past, FP and OOP have sometimes been compared in the context of general methodological problems, like the *datatype extension* *problem* (see e.g. [Rey75, Wad98, KFF98, FF98, Bru03]). *(3) Existence of FP emulation techniques in OOP*: Some common programming practices in OOP include FP-like techniques when solutions are expressed at a function/method level. For instance, "object functions" (*functors*) which encapsulate single methods into objects are often used, e.g. by C++ programmers (see e.g. [SL95, Mey01, Ale01, JF09]). Also, plug-in capabilities, introspection/reflexion, or anonymous class constructs are used for the same reason, e.g. by Java programmers (see e.g. [Blo01, BGS07]) *(4) FP+OOP as a modern trend*: There is nowadays a tendency to propose and include FP-oriented extensions to existing OO languages (in particular to overcome the shortcomings of the above emulation techniques): For instance, several such extensions for C++ were made available (see e.g. [Lau95, MS04]), and C++ standard committees are working on some of them (see e.g. [JFC08, JF09]). Discussions have been held on including FP constructs in upcoming Java versions (see e.g. [Gaf07, Goe07, BGGA08]). Eiffel has included some FP-oriented constructs in its standard – the *agents* [ECM06b, DHM$^+$99]. The C# language also offers FP constructs – the *delegates* – [ECM06a], fully supported since its 3.0 version [Mic07a].

This said, there is one important point to keep in mind: FP offers at least two facets with regard to functions: "*purity*" and "*first-class citizenship*". Purity means mathematical-like functions, without any possibilities of dealing with side-effects: this leads to nice properties, like simpler testing and ready-to-use parallelism. First-class citizenship means functions considered as any other values: this leads to better functional granularity design and specific programming techniques. When blending FP with OOP, one almost always speaks about the latter only, since purity does not have much effectiveness when it is mixed with imperative features. This restricted point-of-view of FP is also the one we take in this paper. Now, even if this kind of paradigm combination has often been mentioned (as the references above show it), it has mostly occurred in a scattered manner, in particular for the statically typed OOP case (see however [Küh99, MS04, Nar05, Mei07]). Accordingly, there is still place for some discussion and synthesis about what programmers could expect from a design methodological viewpoint when FP and OOP are available at the same time. This paper attempts to give such a presentation.

The first section is dedicated to introducing criteria and their consequences to check that some language features make FP-oriented features effective within an OOP context: first-class citizenship properties, closures, mechanisms to define interrelations between FP and OOP, and FP explicit support. This section also introduces one of the central design difficulty when mixing FP and OOP programming techniques: the architectural *design granularity mismatch*, that is, the FP design level, being finer, does not always easily fit with the OOP design level. One must indeed sometimes deal with sets of standalone functions induced by FP in one's OOP architecture. The second section of the paper presents and illustrates a list of general techniques and idiomatic forms available when having FP capabilities within OOP. This encompasses code abstractions at a function/method level, easy iterator-loop programming, operation compositions/comprehensions, and partial applications of functions through currying. The third section discusses the general architectural qualities and drawbacks when using FP techniques as described in the preceding section. The two main qualities are related to a possible reduction of the number of object/class definitions, and to name abstractions at a function/method level. The drawbacks are essentially due to the design granularity mismatch. Common partial solutions to this problem are then discussed, in particular, uses of basic modular (C/Ada-like) components or utility classes, and uses of anonymous constructs. Next, illustrating these facts further, the fourth section analyzes some of the classic OOP design patterns: Strategies and Commands are recalled to be emulations of higher-order functions; Observers, defining mutual relationships between objects through single methods, can be accordingly simplified by FP; Virtual Proxies, being based on an OO evaluation control technique, can be reimplemented through classic FP-oriented *laziness* emulations; Visitors, dealing with functional organizations of sets of methods, may be related to classic FP *data-driven* techniques. These last two patterns allow us to demonstrate how characteristic FP techniques provide OOP with alternative implementation choices. The overall conclusion of the paper will be that FP-oriented features definitely add more possibilities to OOP, sometimes offering simpler and more flexible solutions, but also sometimes at the expense of less architectural and datatype coherence.

Note that the paper will deliberately not include discussions about performance and compilation issues, nor comparisons between existing technical ways of integrating FP into OOP. Our point-of-view will remain here mostly methodological and is expected to be relevant to any OO language with FP capabilities. We have chosen to mainly illustrate the text with C# 3.0 programs, but the examples would essentially take the same form in any functional generic static OO language.

2 EFFECTIVE FP EMBEDDING IN OO LANGUAGES

2.1 Criteria and Rules for FP Embedding

This section presents criteria and properties to check that FP capabilities are truly available in an OOP context, and provide a powerful function/method granularity design level. In this respect, the main criterion is:

Criterion 1 (First-Class Values). *Functions/methods should be* first-class citizens, *i.e. they should have the same properties as simple values like integers: they can (a) be the result of a function, (b) be passed as an argument of a function, and (c) be stored in a data structure*[1].

These properties ensure that a programmer can take full advantage of easy instantiating, transmitting, building, storing functions. To have first-class functions means that one can deal with chunks of code as straightforwardly as with any value: most

[1]When using Properties (a) and (b), functions are said to be of *higher-order*.

of the FP techniques rely on this fact. The practical consequence of Criterion 1 is therefore:

Rule 1 *When Criterion 1 holds, most FP techniques can be applied.*

A natural by-product of the first-class properties is generally to allow functional values to be defined *per se*, without having to name them – that is, as *anonymous functional values* – making their use easier. Note also that in statically typed systems, functional expressions should enjoy some associated *type inference* features so as to be more directly exploitable.

The next criterion is related to what makes functional values work, i.e. **closures**: A closure consists of associating a function code with an environment. This environment is made of the bindings which exist at the time of the function definition, and which are required for the future uses of the function. Thus, first-class properties need closures[2]. To what extent? *Pure FP* is based on first-class *pure functions* (see e.g., [FH88, Rea89]), i.e. functions which behave in a strict independent way from their calling contexts, and whose results depend on their arguments only. Accordingly, pure functions need *complete closures* which include all the bindings, cloned if necessary, into private non-mutable environments. When pure functions are involved, FP induces nice effective properties through *referential transparency*, that is, subsequent function calls with the same arguments always yield the same result: For instance, as already said, one may enjoy simpler testing and ready-to-use parallelism. However, first-class properties do not need complete closures. This is a common situation in imperative-based languages offering FP-oriented features. There, closures are generally restricted to guarantee a good functioning, e.g. by maintaining references without necessarily cloning them. Sometimes also, their private environments are allowed to be mutable. But then, purity easily breaks apart. This leads us to a second criterion to check FP capabilities:

Criterion 2 (Closures). *First-class functions/methods should be implemented as closures, i.e. they should be associated with specific private environments.*

Rule 2 *When Criterion 2 only holds with non-complete closures, most nice properties due to pure FP are expected to be lost. However, FP techniques can still be applied.*

The above two criteria are about FP only, but one must also consider the relationship between FP-oriented constructs and OOP, in particular with respect to the following intrinsic problem: FP and OOP operate on different design granularity levels. Functions/methods mostly belong to the "programming in the small" level, whereas classes/objects/modules mostly belong to the "programming in the large" level, inducing specific code organizations of sets of functions. As a result, the mix of FP and OOP may generate difficulties, which can be summed up into

[2]The word *closure* is sometimes taken as a synonym for *functional value*.

these two questions: *where do we locate the sources of the individual functions in an OOP architecture?* and *how do we relate individual functions to an OOP architecture?* This is what we call here the **FP-OOP design granularity mismatch**. Note that this problem mainly holds in statically typed OO languages where classes are encapsulating units, like in Java, C++, C# , etc.: function definitions due to FP techniques must indeed be located somewhere in one's OO architecture mostly made of encapsulating classes dedicated to describing objects[3]. Taking care of this problem can be associated with a third criterion:

Criterion 3 (FP-OOP interrelation mechanisms)*: Standalone functions/methods should be explicitly relatable to the class/object level.*

Rule 3 *When Criterion 3 holds, it helps solving some of the FP-OOP design granularity mismatch problem.*

Note that a prominent counter-example to the OO "classes-are-also-encapsulating-units" model is given by the Common Lisp Object System (CLOS) (see [GWB91]), where methods live outside classes. For this language, Criterion 3 readily holds since class specifications and implementations mainly rely on the functional level, but at the expense of less modular structure (see e.g. [Kee89]).

Finally, we also add a simple pragmatic fourth criterion, important for a true usability of FP-oriented features in OOP:

Criterion 4 (FP Support)*: The FP-oriented features should be reinforced by related constructs, predefined definitions, occurrences in standard libraries, etc.*

Rule 4 *When Criterion 4 holds, an OOP language acknowledges the fact that FP is one of its fully integrated tools.*

2.2 An Effective FP Construct in OOP: the Delegates

We can now illustrate the above criteria in C# : this language offers a FP-oriented construct called **delegates** (see e.g. [ECM06a, Mic07a, JLT98, Mic98, Ken06]). Here is a simple example illustrating the basics:

```
delegate string StringFun(string s);  //definition of a delegate type

string G1(string s) {           // a method whose type matches StringFun
   return "some string" + s;
}

StringFun f1;                   // declaration of a delegate variable
f1 = G1;                        // method value assignment
f1("some string");              // application of the delegate variable
```

[3]This fact has often been overlooked, in particular when comparing FP and OOP solutions for the datatype extension and expression problems.

By using delegates[4] one can therefore define function types and identifiers acting like any other functions. Note that internally, C# delegates are just represented as a special kind of objects[5] (FP-oriented constructs in OOP do not necessarily imply deep language extensions). Thus, being objects, delegates directly satisfy the basic first-class properties. Here are some examples demonstrating this fact (`StringFun` and `G1` are defined as above):

1. Delegate types can be used for method parameters, and delegates can be passed as arguments of methods as any other values:

   ```
   string Gf1(StringFun f, string s) { [...] }  //delegate f as a parameter
   WriteLine(Gf1(G1, "some string"));            //call
   ```

2. Delegates can be returned as the result of a computation of a method. For instance, assuming `G2` is a method acting on strings and defined in the class `SomeClass`:

   ```
   StringFun Gf2() {                    //delegate as a return value
      [...]
      return (new SomeClass()).G2;
    }

   WriteLine(Gf2()("some string"));   //call
   ```

3. Delegates can occur in data structures:

   ```
   var l = new LinkedList<StringFun>();        // list of delegates
   l.AddFirst(G1);              // insertion of a delegate in the list
   WriteLine(l.First.Value("some string"));   // extract and call
   ```

As expected, delegates may also become standalone values, that is, they can be defined *anonymously*, and can directly instantiate variables and parameters:

```
delegate(string s) { return s + "some string"; };
```

In C#, such an anonymous delegate definition may even look more like a usual *lambda expression* (or to a *block with arguments* in Smalltalk terminology [GR89]):

```
(s => { return s + "some string"});
s => s + "some string";  // equivalent syntax
```

[4]The word *delegate* seems to clash with the word *delegation* commonly used in OOP design, that is, dynamic compositions of objects where method calls are forwarded onto other objects (see e.g. [Lie86, GHJV95]). One can also find some similarity at a more basic level: C# delegates, representing functional values, also defer to runtime the decision about which methods should actually be called (see also Sec. 4.2). Note that in this respect, C# delegates are often said to be similar to C/C++ function pointers and associated with *callbacks*. In fact, being object-like and type safe values, delegates are more similar to C++ *functors* than to function pointers (ad-hoc FP-oriented extensions of C++ are generally based on templatized functors – see e.g. [Lau95, MS04]).

[5]A C#-delegate declaration implicitly defines a new class derived from the predefined class `System.Delegate`. Delegates are instances of this class, and they are able to encapsulate a single method (or a sequence of methods). For instance, `StringFun f1 = G1` is the same as `StringFun f1 = new StringFun(G1)`. The specific behavior of such an object is then the following: given an appropriate set of arguments (with a functional syntax), its encapsulated method is directly invoked with these arguments (see e.g. [Ric01]).

C# does not define a pure FP context, and as a consequence, closures are not complete. For instance, consider the following definitions:

```
StringFun f1, f2;
int counter = 1000;
f1 = s => s + counter.ToString();
f2 = s => s + counter.ToString();
```

Here, `counter` is shared between `f1` and `f2`. Any modification of `counter` has an effect on `f1` *and* `f2`: In C# closures, only references to outer values and objects within a delegate are preserved, so that the delegate functioning is guaranteed, but not its independence to the calling context. When necessary, a classic technique to overcome this weakness is to use the lexical scope semantics: One captures parts of a local private environment while building a function. For instance:

```
StringFun F() {
   int counter = 1000;
   return s => { return s + counter.ToString(); };
}
```

In this example, each delegate obtained through a call to `F()` contains an individual private `counter`. Note that this technique has been sometimes applied to emulate classes in FP, when OOP constructs are not available (see e.g. [SF90, FWH92] for the Scheme case).

Added to the above key FP-oriented properties (see Criteria 1 and 2), some features of C# encourage the use of delegates, thus promoting FP (see Criterion 4):

- *Basic Delegate Predefinitions.* C# offers predefined functional and procedural generic delegate types (respectively under the name `Func` and `Action` – overloading applies to generic delegate type names too):

  ```
  delegate TResult Func<TResult>();
  delegate TResult Func<T,TResult>(T a1);
  delegate TResult Func<T1,T2,TResult>(T1 a1, T2 a2);
  delegate void Action();
  delegate void Action<T>(T a1);
  etc.
  ```

- *FP Uses in the Standard Library.* Some of the basic functions provided by the .NET Framework are higher-order functions, e.g., comparators, sorters, iterators, filters, etc.

- *First-Class Multiple Invocation and Multicasting.* A C# delegate may itself contain an *invocation list* of delegates, that is, an ordered set of delegates. When such a delegate is called, all the delegates in its invocation list are called in order. The result value is determined by the last method called. As a consequence, sets of delegates may easily be composed, and still remain first-class values. Of course, since only the last method result is taken into account, this technique mainly applies to imperative-oriented uses with side-effects like e.g., *object multicasting* (see the examples further in Sec. 5.2). Note also that the syntax to manage insertions and deletions in such delegate invocation lists is simplified by the overloading of the operators +, -,+=, and -=.

- *Function Marshalling and Serialization.* C# allows lambda expressions to be represented as data structures called *expression trees* (see e.g. [Mic07a, Wie08]). As such, they may be stored and transmitted.

2.3 An Interrelation FP/OOP: the Extension Methods

We defined Criterion 3 as the possibility of relating single functions/methods to the overall OOP architecture. C# offers such a mechanism called **extension methods**. It allows programmers to add methods to existing classes without creating new derived classes. These extension methods may be invoked as if they were static methods of the extended classes. For instance, the following extension method of the class String counts the words separated by blanks in a string (the first parameter of such a method specifies which class the method extends, and must be preceded by a "this" keyword):

```
static int SimpleWordCount(this String str) {
   return str.Split(new char[] { ' ' }).Length;
}
```

This method can then be directly used by an instance of String:

```
String s1 = "some chain";
s1.SimpleWordCount();    // usable as a String method
SimpleWordCount(s1);     // also usable as a standalone method
```

An extension method can also be defined for interfaces, and therefore at the roots of class hierarchies. Here is a C# classic instance of a generic extension method:

```
static IEnumerable<T> MySort<T>(this IEnumerable<T> obj)
   where T : IComparable<T> {
   [...]
}

List<int> l = [...];  // List is a subtype of IEnumerable
l.MySort();
```

Extension methods provide a mechanism to easily extend classes and to connect isolated methods to existing OOP architectures. This mechanism integrates some of the "*open class*" concept into the language (see e.g. [CMLC06]), and increases its "*operation-centric dimension*" (as defined in [GWB91] for CLOS). However, C# extension methods cannot access private data of the class they are extending, so that they cannot serve as a mean to violate object encapsulation. Moreover, they are not polymorphic – there is no dynamic selection over them – since they are bound at compile-time, and they must be defined within static classes. Thus, their ability to directly solve specific OOP architecture problems is limited (see e.g. Sec. 5.5). Nevertheless, extension methods can be used to connect delegate implementations to existing classes, and as such, they help building explicit bridges between FP and OOP.

Note that other kinds of such interrelation mechanisms have been proposed. One of them is to make effective the natural relationship between single methods and classes with only one method. For instance, *closure conversions* have been suggested for Java [BGGA08]: roughly, every functional value of type t may be cast to any class type/interface that declares a single method with type s, if t is a subtype of s.

3 GENERAL FP TECHNIQUES IN OOP

This section discusses in a general setting what are the programming techniques and consequences of having FP capabilities in OOP, making a functional granularity fully available within OOP.

3.1 Code Abstraction at a Function/Method Level

A first immediate consequence of first-class functions is the possibility of applying *separation of concerns* at a function level [Hug89]: any specific behavior, any snippet of code within a function/method can be abstracted out into a functional parameter. In other words, one can easily define generic functions which act as "pattern templates" at a function/method level. For example, consider some method involving some specific code:

```
float M(int y) {
   int x1 = [...]; int x2 = [...];
   [...]
   [...code...]; //some code using x1, x2, y
   [...]
}
```

Assume that `M` has to be modified with respect to `[...code...]` and some of its local data. Using first-class functions, these actions can be factored out into a functional parameter `f` as follows:

```
public delegate int Fun(int x, int y, int z);

float MFun(Fun f, int x2, int y) {
   int x1 = [...];
   [...]
   f(x1, x2, y);
   [...]
}
```

Next, assuming the existence of `F1` and `F2` which satisfy the functional type `Fun`, different versions of `MFun` can be obtained and applied:

```
int z1 = MFun(F1, 1, 2);
int z2 = MFun(F2, 3, 4);
```

3.2 Generic Iterator and Loop Operations

A celebrated application of the above code abstraction technique is the generic implementation of uniform and iterated transformations of data structure (see e.g. [BW88]). For instance, *map*-like functions, – usually called *internal iterators* – take a function f and a data structure instance D as arguments. The function f is then called for each element of D, and returns a new instance of the same kind of data structure as D made of the results of these calls. Here is a simple *map* implementation for lists in C# (using the predefined `Func` delegate type):

```
List<T2> Map<T1,T2>(this List<T1> data, Func<T1, T2> f) {
   List<T2> res = new List<T2>();
   foreach (T1 item in data) { res.Add(f(item)); }
   return res;
}
```

With anonymous functional expressions, this method can be called as in any FP language. For instance, here is how to square every element of a list:

```
someList.Map( i => i * i );
```

Predefined *map*-like functions are available in the .NET framework: `Map` which is essentially defined as above, and `App` which iteratively applies a procedure p to every element of a data structure D. In the same vein, $fold$-like or *reduce*-like functions transform data structures into single values. For instance, a fold over lists takes a binary function as an argument, called in turn for each element of the list and each intermediate value due to the current state of the folding, and finally produces a single value. Here are two simple examples of folds – called `Aggregate` in C# – to respectively compute the sum and the product of the elements of a list:

```
int sum = someList.Aggregate((accum, x) => accum += x));
int prod = someList.Aggregate((accum, x) => accum *= x));
```

These iterator-oriented functions generally lessen the burden of managing explicit ad-hoc iterator classes, without impairing power and flexibility (see also e.g. [Bak93, Küh97, Gaf07, Bis08]).

3.3 Operation Compositions and Sequence Comprehensions

First-class functions/methods may also provide convenient ways of expressing sequences of operations as compositions of higher-order functions. Consider the following example (inspired from [Coc08]):

```
public static void PrintWordCount(string s) {
   String[] words = s.Split(' ');
   for (int i = 0; i < words.Length; i++)
       words[i] = words[i].ToLower();
   var dict = new Dictionary<string, int>();
   foreach (String word in words)
       if (dict.ContainsKey(word))
           dict[word]++;
```

```
        else dict.Add(word, 1);
    foreach (KeyValuePair<String, int> x in dict)
        WriteLine("{0}: {1}", x.Key, x.Value.ToString());
}
```

This piece of code is nothing but a pipe over arrays of strings. In order to improve its qualities, one can factor it out into a sequence of auxiliary methods, some naturally becoming higher-order iterator-like operations:

```
public static void PrintWordCount(string s) {
    String[] words = s.Split(' ');
    String[] words2 = (String[]) Map(words, w => w.ToLower());
    Dictionary<String, int> res = (Dictionary<String, int>) Count(words2);
    App(res, x => WriteLine("{0}: {1}", x.Key, x.Value.ToString()));
}
```

But, intermediate variables still clutter the code. One could then consider using generic higher-order composition functions like:

```
Func<T1, T3> Compos<T1, T2, T3>(Func<T1, T2> f, Func<T2, T3> g) {
    return t => g(f(t)); }
```

However, by defining the above auxiliary methods as *extension methods*, there is a simpler FP-oriented way in C# of expressing the same idea:

```
public static void PrintWordCount(string s) {
    s.Split(' ')
     .Map(w => w.ToLower())
     .Count()
     .App(x => WriteLine("{0}: {1}", x.Key, x.Value.ToString())); }
```

The operation composition is now obtained as a *sequence of method calls* (the dot "." becomes the composition operator), and the pipe-like form of the process clearly appears. Moreover, because of its streaming-like evaluation tactics in C#, this technique can be related to *list comprehensions* as they are available in Haskell. This idea has been exploited further in the *Language Integrated Query* (LINQ) where a set of higher-order functions are defined to easily express "*query comprehensions*". This library simplifies the relationships between OOP and relational data, and accordingly reduces the classic "*OO-relational impedance mismatch*" problem [Mei07]. For instance, here is how may look such a query:

```
var query = giraffes
            .Where(g => g.Age > 5)
            .OrderByDescending(g => g.Age)
            .GroupBy(g => g.NeckLength > 10);
```

This FP-oriented composition technique definitely increases readability and code compactness. In the general case, however, there may be some inconveniences: First, composing higher-order functions in such a dense way within a non-pure FP context – i.e. with side-effects – may be subject to difficult testing and bug tracking (in this respect, note that LINQ has been designed to limit these risks by integrating pure FP into it as much as possible – see e.g. [Mei07]). Also, since this technique is here based on extension methods, if it is applied to too many ad-hoc situations, it may wildly multiply the links between classes, and thus the OO architecture complexity.

3.4 Function Partial Applications and Currying

There is another general programming technique available along with basic FP capabilities: *function partial application*. Indeed, with higher-order functions, every n-ary function can be transformed into a composition of n unary functions, that is, into a *curried function* (see e.g. [FH88, Rea89]). For instance, consider in C# a simple function in its lambda expression form which adds two integers:

```
Func<int, int, int> lam1 = (x, y) => x + y;
```

A call to this functional value can be for instance `lam1(3, 4)`. The equivalent curried form of `lam1` is then the following:

```
Func<int, Func<int, int>> lam2 = x => (y => x + y);
```

The same call as above becomes `(lam2(3))(4)` (which can even be simplified into `lam2(3)(4)`). The advantage of such a curried form is that one can fix the first parameter and still obtain a value which is a function over only one parameter:

```
Func<int, int> lam3 = lam2(3); //partial application
```

The functional value `lam3` is now equivalent to the unary function "`y => 3 + y`". Such function partial applications are quite handy since they allow one to delay the instantiations of some of the parameters of a function: one may thus take advantage of specific environments to specialize a function in an ad-hoc manner, or to reduce its number of parameters to comply to existing function types. In typed FP languages like Haskell and ML, but also in cross-paradigm languages like Scala and OCaml, one may directly express functions in their curried form (this is not the case in C#).

A classic FP exercise is to obtain higher-order functions able to transform non-curried functions into curried ones. For instance, here is a generic C# implementation for binary functions (defined as an extension method of `Func` so that its calls become available with a class notation, i.e. it becomes composable with the dot "." notation):

```
public static Func<T1, Func<T2, TRes>>
   Curry<T1, T2, TRes> (this Func<T1, T2, TRes> f) {
      return (x => (y => f(x, y)));
}

Func<int, int> lam4 = lam1.Curry()(3);  //partial application
```

4 FUNCTIONAL GRANULARITY DESIGN IN OOP

In the preceding section, we have listed up the main techniques that basic FP makes possible in OOP. These were essentially "programming in the small" techniques, mostly relying on functional code abstraction. We now develop what are their qualities and drawbacks when they are used within some OOP architectural design.

4.1 Reduction of the Number of Object/Class Definitions

An important consequence of having FP capabilities is that one can avoid some class definitions. Indeed, reconsider the method M of Sec. 3.1 and its transformation into MFun: In a pure OOP approach, the same situation is generally solved by using an object-based abstraction, often called *function object* or even *functionoid* (see e.g. [SL95, Küh97, Mey01]). First, an interface – call it IFun – is needed to describe the type of the objects dedicated to encapsulate the abstracted method f. Second, distinct classes derived from IFun are needed for every distinct implementation of f, e.g., F1, F2. Finally, instances of IFun can be transmitted as regular objects:

```
interface IFun{
   int F(int x, int y, int z);
}
class F1 : IFun {
   public int F(int x, int y, int z) { [...] }
}
class F2 : IFun {
   public int F(int x, int y, int z) { [...] }
}

float MObjFun(IFun funobj, int x2, int y) {
   int x1 = [...];
   [...]
   funobj.F(x1, x2, y);
   [...]
}

int z1 = MObjFun(new F1(), 1, 2);
int z2 = MObjFun(new F2(), 3, 4);
```

In comparison, the FP solution only requires a new delegate type definition, which may belong to the class where it is used: it clearly avoids cluttering the global architecture with new classes (see also e.g., [Bec97]).

Note however that an OOP solution with *function objects* as above makes possible more than just method encapsulation, since function objects may also include their own data, methods, etc. Here is an example using function objects including their own constructors. Consider the following class also satisfying the interface IFun:

```
class F3 : IFun {
    int state;
    public F3(int a) { state = a; }
    public int F(int x, int y, int z) { [... state ...] }
}

int z3 = MObjFun(new F3(10), 3, 4);
```

Here, F3 has an attribute state which is used in the body of F, and which is set by the constructor F3(int a). The parameter “a” can then be considered as a parameter of F too: the difference with the other parameters of F is that “a” is fixed when instantiating F3. As a consequence, we obtain a kind of *partial application* of the method F. Note also that F may now be considered as a function having a different signature than the ones defined by F1 and F2, while still satisfying IFun and still preserving type-safety. This technique of partial application and variability of function types has been described e.g. in [Küh94, Cli06], and it is the OOP counterpart of the currying functional technique introduced in Sec. 3.4.

4.2 Name Abstraction at a Function/Method Level

Whenever a variable occurs in one's code, the origin and the initial name of what it denotes does not matter. With FP, this rule applies to functions too (see e.g. [Hud00]). For instance, consider the following C# code:

```
delegate string StringFun(string s);
StringFun f1;
[...] f1 [...];
```

Here, `f1` is just a placeholder for any method of type `StringFun` whatever its origin and initial name are. On the contrary, in OOP, denoting a single method is obtained by calling it through some object: As a consequence, name abstraction is effective only at the object level, not at the function/method level (see also [Küh94]). For instance, consider the above code rewritten in OOP style:

```
interface IStringFun{ string F1(string s); }
IStringFun obj1 = [...];
[...] obj1.F1 [...]
```

Here, `obj1` is a placeholder for any object obtained through a class derived from `IStringFun`, but `F1` is a name which *must* be used.

Let us illustrate this name abstraction effect with a less formal example. Consider the case of an *object composition-delegation* in a classic OO Bridge design pattern [GHJV95]:

```
public class Window {
   private WindowSys _imp;
   void DrawFigure ([...]) {
      _imp.DeviceFigure([...]);
   }
}
```

The above code is flexible from the point-of-view of the object denoted by `_imp`, but it must stick to the fact that the object `_imp` has a method `DeviceFigure` declared in the class `WindowSys`. When method names do not fit, the usual OOP solution is to implement Adapter classes like:

```
class Adapt : WindowSys {
   [...]
   void DeviceFigure([...]) {
      adaptee.SomeOtherMethodName([...]);
   }
}
```

Objects of class `Adapt` may now instantiate the attribute `_imp` in `Window`. This kind of adapters often occurs in pure OOP as instantiations of inner anonymous classes to directly assign compatible objects (see e.g. the idiomatic uses of the

ActionListener class in Java). It can also be generalized by reflection/introspection-based solutions (similar to the *Pluggable Selector* Smalltalk pattern [Bec97, BGS07]). But using the function/method granularity level simplifies the situation and reduces the syntactic overhead:

```
public delegate void DeviceFigureFun([...]);

public class Window {
   private DeviceFigureFun _devfigure;
   void DrawFigure ([...]) {
      _devfig([...]);
}
```

Here, any method satisfying DeviceFigureFun is entitled to instantiate _devfigure.

Such name abstraction at a function level can also be related to the distinction between *nominal* and *structural* compatibilities (see e.g. [Bru02, Ost08]). Recall that in the nominal case, type compatibilities depend on explicit declarations, generally by explicit class inheritances and subtyping definitions. In the structural case, only the types of the objects/values are considered to induce compatibilities. Structural relationships make unanticipated reuses possible, and limit multiplication of similar type definitions, whereas nominal relationships allow one to explicit design intents and to express tighter contracts. Most of the mainstream OOP languages are nominal at their object/class design level[6]. Nevertheless, typed first-class functions can be seen to provide some structural flavor within the nominal case: name abstraction at a function level indeed means that as soon as a function satisfies the declared type of a parameter or a variable, it can instantiate it whatever its origin is.

4.3 Design Granularity Mismatch

According to the preceding sections, FP capabilities appear quite effective for OOP. However, the discussion must also include the *design granularity mismatch* (see Sec. 2.1). For instance, consider again the formal higher-order function MFun in Sec. 3.1 (developed further in OOP in Sec. 4.1), that is:

```
public delegate int Fun(int x, int y, int z);
float MFun(Fun f,...) { [...] }
```

For every F1, F2,... satisfying the type Fun, the method MFun can be called. But questions arise: *Where do we locate the implementations of the* Fi*'s in one's architecture which happens to be mostly OO?* and *how do we relate the* Fi*'s to this OO architecture?* Moreover, the set of the Fi's is expected to change and grow: according to the idea of function abstraction, the Fi's are indeed a variable part of the code. Therefore, one may also ask: *how do we organize and collect the implementations of the* Fi*'s?*. Of course, if the Fi's are methods already belonging to some classes or objects, or if they are the results of some computation, there is no architectural problem; if not, programmers must certainly find answers to the

[6]Nominal compatibility is however not intrinsic to OOP (see e.g., the object system of OCaml [LDG$^+$08]).

above questions. We have already seen that explicit interrelation mechanisms like *extension methods* give some answers to the second question (see Sec. 2.3). However, nothing has been said yet about how and where to put the Fi's codes. There exist two main general solutions to this problem, both showing some inconveniences:

- **Solution with Basic Modules or Utility Classes**: One may place the Fi's implementations as static methods in some plain module or some utility class. This technique amounts to including or emulating basic (C/Ada-like) modular components into an OOP architecture, for instance in the following form in C#:

```
class U {
   static [...] F1 [...];
   static [...] F2 [...];
   [...]
}
```

From an organization point-of-view, this technique means collecting the different functional cases – the Fi's – into single modular components, whereas in the usual OOP style, they are spread out over different existing classes depending on a same root class (see e.g. [Rey75, FF98, KFF98]). Accordingly, these two different function organizations are sometimes respectively called *functional decomposition* and *OO decomposition*. Mixing these two techniques may be acceptable, but it certainly increases the complexity of one's architecture: modular units describing objects (the classes) are associated with others only describing plain sets of functions. Moreover, basic modules or utility classes are difficult to evolve. In particular, adding some new functions to the existing Fi's may induce awkward refactoring moves: either by directly adding them to the module/class containing the Fi's, or by using inheritance or generics, generating cumbersome and not so justified class hierarchies (see also the discussion further about Visitors in Sec. 5.5).

- **Solution with Anonymous Constructs**: Alternatively, one may use anonymous constructs, that is, one may directly provide the needed implementations along with the calls to the higher-order elements. These calls become then self-sufficient. For instance, the different calls to MFun in Sec. 3.1 can be expressed as:

```
int z1 = MFun((x1, x2, x3) => [...F1 code...], 1, 2);
int z2 = MFun((x1, x2, x3) => [...F2 code...], 3, 4);
```

instead of:

```
int z1 = MFun(F1, 1, 2);
int z2 = MFun(F2, 3, 4);
```

Such anonymous constructs clearly solve the problems generated by the solution using basic modules, since one spares the trouble of explicitly inserting new components into one's architecture (see e.g [HT06, JF09]). Moreover, having both anonymous classes and functions allows programmers to choose the right granularity design level for each call (see e.g. [JLT98, Mic98, Gaf07]). However, even if this technique may sometimes work well, anonymous constructs induce a spread of the code over all the calls including them, and this code happens to be not directly reusable or extensible (anonymous constructs are generally associated with a unique use).

Let us sum up the situation: when FP techniques are mixed with OOP, definitions and implementations of individual functions must take their place somewhere in the architecture. This may not be easy in OO languages like Java, C++, C# , where almost everything is defined into classes being encapsulating units and defining object types. Functions definitely belong to a finer level. One may apply the above solutions, but one may also sometimes expect some shortcomings. Of course, in OO languages where methods are not necessarily contained in classes, the design granularity mismatch does not occur in the same proportion, but as already said, at the expense of the modular structure quality.

5 SOME OO CLASSIC DESIGN PATTERNS WITH FP

In the preceding pages, we sometimes mentioned classic OO design patterns [GHJV95] to illustrate FP techniques in OOP. In fact, some of these patterns happen to be themselves closely related to FP techniques (see e.g. [Küh99, MS04, Nar07]).

5.1 Strategy

The intent of a Strategy pattern is to *let an algorithm vary independently of clients that use it*[7]. This flexibility is based on *object delegation* and it is expected to be dynamic (on the contrary, the Template pattern is based on inheritance). In other words, a Strategy relies on abstractions at a function level, and on passing around single function objects at runtime. A classic OO Strategy implementation therefore requires some object encapsulation of functions, as well as some interface to describe their type (see the example in Sec. 4.1). FP and its higher-order functions make this pattern immediate and spare the effort of developing new class hierarchies. Moreover, one can also exploit the functional name flexibility (see Sec. 4.2): any function with the right type may serve in a Strategy, notwithstanding its origin.

The benefits of the FP-oriented Strategy were discovered a long time ago, being directly related to functional abstraction. For instance, Smalltalk programmers are used to exploiting it (see e.g., the *Pluggable Block* pattern [Bec97]). Also, since C# 3.0 integrates FP as one of its acknowledged tool, one finds collections of predefined FP-oriented Strategies in its standard library. For example:

```
public delegate int Comparison<T>(T x, T y)
public void Sort(Comparison<T> comparison)
```

Note that the State pattern which *allows an object to alter its behavior when its internal state changes* can be seen as a generalized version of a Strategy. Accordingly, when a State is reduced to single methods, it may also take advantage of such a FP-oriented implementation (see e.g. [Bis08]). Of course, one must not forget here about the drawbacks of a FP granularity design embedded into OOP (see

[7]In this section, pattern intents are taken from [GHJV95].

Sec. 4.3): organizing sets of standalone functional Strategy instances within one's overall architecture may be difficult.

5.2 Command

The Command pattern *encapsulates requests – method calls – as objects* so that they can easily be transmitted, stored, and applied. This is just an OO way of speaking about encapsulated *callbacks*. From an OO architecture design point-of-view, Commands are similar to Strategies except that their purpose is to carry autonomous procedures instead of functions. Accordingly, using first-class procedures may also simplify the implementation of Commands (see e.g. [SM00, Met04, Bis08]).

A classic example of the usefulness of FP-oriented Commands applies to the development of flexible interactive programs like menus. Menus must be able to easily evolve, and they must be able to fire up actions without having to depend on the initial names or origin of these actions. A function/method design level with FP capabilities directly provides these properties. The C# standard library offers classes to help building such interactive programs based on FP-oriented Commands. In particular, there is a predefined delegate type for Command-like values:

```
public delegate void EventHandler(Object sender, EventArgs e)
```

For instance, the predefined class `MenuItem` contains an attribute `Click` of type `EventHandler`, and `Click` is called whenever the menu item is clicked or selected using a shortcut or an access key. One can then associate a click behavior simply by method assignment. For instance (see [Mic07b]):

```
private void MenuBehavior_Click(object sender, System.EventArgs e) {
   OpenFileDialog fd = new OpenFileDialog();
   [...]
}

public void CreateMyMenu() {
   MenuItem menuItem1 = new MenuItem();
   [...]
   menuItem1.Click = MenuBehavior_Click;  //method value assignment
}
```

Such a delegate assignment is in fact commonly expressed by using anonymous delegates (see also Sec. 4.3):

```
menuItem1.Click = (sender, e) => {
                      OpenFileDialog fd = new OpenFileDialog();
                      [...]
                    }
```

Also, built-in multicasting (see Sec. 2.2) is frequently used in this context, and the exact C# idiomatic form of this kind of assignments is the following:

```
menuItem1.Click += (sender, e) => {[...]};
```

Note that even if FP-oriented first-class Commands lead to a quite effective programming technique – and as such, it is commonly adopted by the C# programmers –, it is far from the pure FP world, being essentially of imperative nature.

5.3 Observer

The Observer pattern intent is also a very general one: it *defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.* In its simplest form, this pattern involves a *subject* (an "observable" component) and multiple *observers* of this subject. Observers are considered as *attached* to the subject: when the subject changes its state, it *notifies* its observers such that they can be *updated*. FP capabilities can simplify this pattern with respect to the way notifications and updatings are managed: these can indeed be represented by lists of first-class methods (see also [SM00, Met04, HT06, Bis08]). For instance, consider first the core implementation of a simple generic OO Observer:

```
public interface Observer<S> {
   void Update(S s);
}

public abstract class Subject<S> {
   private List<Observer<S>> _observ = new List<Observer<S>>();

   public void Attach(Observer<S> obs) {
      _observ.Add(obs);
   }
   public void Notify(S s) {
      foreach (Observer<S> obs in _observ) {
         obs.Update(s);
      }
   }
}
```

Here, each instance of `Subject` maintains a list of its linked observers in order to be able to call the respective `Update` methods of each of these observers. Using FP, this implementation can be simplified by just maintaining a list of the updating methods. In C#, one may also exploit built-in multicasting for this purpose:

```
public delegate void UpdateFun<S>(S s);

public abstract class Subject<S> {
   private UpdateFun<S> _updateHandler;

   public void Attach(UpdateFun f) {
      _updateHandler += f;
   }
   public void Notify(S s) {
      _updateHandler(s);
   }
}
```

As expected, one of the architectural consequences of this FP-oriented version is that observers do not need any specific method called `Update()`: they just make use of the functional name abstraction technique (see Sec. 4.2). The interface `Observer` is not even necessary any longer. Thus, like for the previously discussed design patterns, the FP-oriented implementation relaxes some of the architectural overload induced by a strict OOP implementation. Again, however, there may be problems to organize and include individual update functions in the overall architecture.

5.4 Virtual Proxies and Laziness

There are programming situations where a general optimization technique can be fruitfully applied: delaying the evaluation of function calls until their results are actually needed. This technique can be summed up by the motto "don't compute anything unless you have to", and it is related to the classic evaluation tactics named *call-by-need* or *lazy evaluation* (see e.g. [FH88, FWH92]). In OOP, this idea may be implemented through Virtual Proxies, which are objects that *provide surrogates or placeholders for other objects so that their data are created/computed only when needed*. Generally, laziness is also associated with *memoization*, that is, capturing/storing computed data so that they are computed only once. Virtual Proxies often include this idea too.

Lazy evaluation is one of the distinctive features of pure FP, being directly applicable because of *pure functions*, and more generally because of *referential transparency* (see Sec. 2.1). In non-pure FP, when *eager evaluation* holds, emulation of lazy evaluation is possible and is based on the following interpretation (see e.g. [Rea89, FWH92, Pau97]): a function encapsulates an expression – its body –, which is evaluated only when the function is called. Thus, when functions are first-class values, expressions can be transformed into first-class values whose results are obtained only when needed, i.e. when they are called. In typed FP, the signature of such lazy functional values is commonly "`() => T`", that is, a function without any argument and a result of type `T` (the type of the lazy value). In C#, this type is predefined as `Func<T>`, and lazy values with memoization can be implemented as follows (see also [Pet07]):

```
public class Lazy<T> {
   private Func<T> _lazyExpr;
   private T _value;
   private bool _immediateValue;  //explicit flag for memoization

   public Lazy(Func<T> e) {
      this._lazyExpr = e;
      _immediateValue = false;
   }
   public T Value {
       get { if (!_immediateValue) {
                 _value = _lazyExpr();  //evaluation
                 _immediateValue = true;
             }
             return _value;
       }
   }
}
```

Objects obtained from this class are then used as follows:

```
var lazyval1 = new Lazy<int>(() => { return [...]});
int i = lazyval1.Value;
```

Emulated laziness can also be more directly implemented at a FP level. For this purpose, one uses a function to transform a plain expression of type "`() => T`" into a lazy value associated with a memoization tactics. Here is such a function based on the closure capturing technique (as seen in Sec. 2.2):

```
public static Func<T> LazyF<T>(this Func<T> e) {
   T _value = default(T);
   bool _immediateValue = false;
   return () => {
      if (!_immediateValue) {
         _value = e();  //evaluation
         _immediateValue = true;
      }
      return _value;
   };
}
```

Evaluation control can now be applied without using any classes:

```
var lazyval1 = LazyF<int>(() => { return [...]});
int i = lazyval1();
```

These laziness emulations can be exploited whenever evaluation control is needed (see e.g. [Rea89, Pau97]), e.g. to build lazy data structures, to implement sequence comprehensions, and also to help implementing Virtual Proxies. For instance, here is a simple situation where "large computations" are assumed to be involved when objects are instantiated:

```
public interface IProc { void Process(); }

public class Simple : IProc {
   private int _state;

   public Simple(int i) {
      _state = SomeClass.LargeComputation(i);
   }
   public void Process() { [...] }
}
```

A classic OO Virtual Proxy implementation delaying this large computation until the associated instances are actually needed is the following:

```
public class SimpleProxy : IProc {
   private Simple _simple = null;
   private int _arg;

   public SimpleProxy(int i) {
      _arg = i;
   }
   protected Simple GetSimple() {
      if (_simple == null)
          _simple = new Simple(_arg);
      return _simple;
   }
   public void Process() {
      GetSimple().Process();
   }
}
```

Using emulated laziness, the implementation of such a Virtual Proxy can be simplified. The evaluation control mechanism of the Virtual Proxy can indeed be factored out by emulated laziness, and can be uniformly applied in the whole architecture:

```
public class SimpleLazyProxy : IProc {
   private Func<Simple> _simpleLazy;

   public SimpleLazyProxy(int i) {
        _simpleLazy = LazyF<Simple>(() => new Simple(i));
   }
   public void Process() {
      _simpleLazy().Process();
   }
}
```

Note that emulated laziness has some drawbacks too: for instance, its lazy values are not directly available, and they must always be explicitly applied. Note also that all the elementary operations over them must be reimplemented accordingly.

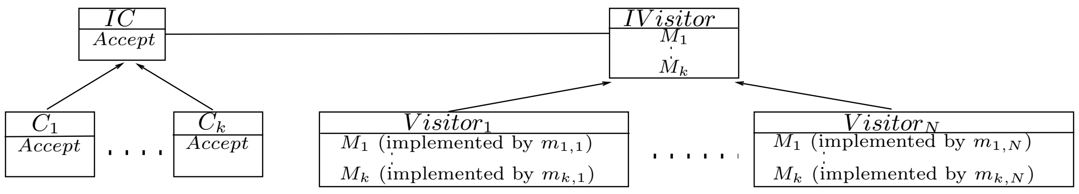
5.5 Visitor and Data-Driven Programming

Let $\{C_i\}_{i=1,..,k}$ be a set of classes which must be extended by some new method, inducing k distinct implementations (one for each C_i). Consider the situation where this kind of extension occurs N times, so that we end up with $k \cdot N$ different method implementations, denoted by $m_{i,j}$; the jth extension of the C_i's is denoted by $Extend_j = \{m_{i,j} | i = 1, .., k)\}$. *A priori*, each C_i must be derived N times, leading to $k \cdot N$ different classes (one for each $m_{i,j}$), and therefore, to a wild OOP architecture. The Visitor pattern offers a solution to this problem as it makes a class hierarchy more easily extensible, in particular when classes are strict encapsulating units[8]: *A Visitor lets you define new operations without changing the classes of the elements on which they operate.* Recall indeed that the classic OOP way of implementing Visitors is the following:

1. One creates a Visitor interface $IVisitor$ including k signatures M_i which correspond to the generic methods assumed to extend each class in $\{C_i\}_{i=1,..,k}$.

2. For each extension set $Extend_j$, one creates a concrete visitor class $Visitor_j$ deriving from $IVisitor$. This class contains the implementations $m_{i,j}$ of the M_i's, for $i = 1, ..., k$.

3. One makes the classes C_i able to use Visitor instances of the classes $Visitor_j$'s, and able to select the right implementation $m_{i,j}$ from these instances. This last step is generally implemented by explicit *double-dispatch* over the $m_{i,j}$'s with methods called *Accept*.



[8]The Visitor pattern loses some of its relevance in languages in which *open classes* are available.

Now, even if the Visitor design pattern is a quite powerful idea, it also brings several well-known drawbacks. Among them:

- *Refactoring Resistance.* A Visitor definition depends on the visited classes C_i on which it operates. When refactoring/extension moves are applied to the C_i's, the whole Visitor class hierarchy must be edited – for instance, when a new class is added to the C_i's.
- *Staticness.* A Visitor has an important static part since the concrete visitors $Visitor_j$ rely on class derivation. This is an advantage in terms of type-safety, but it also limits flexibility – for instance, the $m_{i,j}$'s are fixed at compile time.
- *Invasiveness.* A Visitor requires that the visited classes C_i anticipate and/or participate in making the selection of the right method $m_{i,j}$ – for instance, by using *Accept* methods and double-dispatch tactics.
- *Naming Inflexibility.* A Visitor requires that all the different implementations of $m_{i,j}$ be uniformly named M_i so that they can be uniformly invoked by the visited classes C_i. This constraint may limit flexibility and induce code duplication problems (see also the discussion about name abstraction in Sec. 4.2).

Refactoring resistance is considered as the main problem of the Visitor pattern (see e.g. [GHJV95, Ale01]). In fact, this problem may be related to the discussion about the design granularity mismatch in Sec. 4.3, and its solution using basic modules and utility classes to collect implementations of functions. Indeed, a concrete visitor $Visitor_j$ collects the implementations of one single extension of all the C_i's: this is a functional-like organization. Concrete visitors are often utility classes. More generally, Visitors are indeed a way of mixing *OO decompositions* with *functional decompositions*. This question has been also more generally discussed with respect to datatypes (see e.g. [Wad98, KFF98, Bru03]).

Now, since functional decomposition inherently exists in a Visitor, FP can be expected to induce alternative ways of implementing this design pattern. First, note that when staticness and type-safety can be relaxed, some FP-oriented OOP solutions of the Visitor have been designed to alleviate the *refactoring resistance* (see e.g. the *acyclic Visitor* [VMR98, Mar03] for OO solutions using mostly sets of *function-objects*, and see [PJ98] for OO solutions using class introspection/reflection techniques). Also, language constructs like *extension methods* (see Sec. 2.3) can provide straightforward solutions: one just externally adds the methods $m_{i,j}$'s to the C_i classes by defining them as extension methods. This technique corresponds to what is possible in OO languages offering *open classes* like CLOS [GWB91], or MultiJava [CMLC06], but at the expense of the quality of the modular architecture structure. Note that two things prevent such technique to be effective in C#: extension methods in this language rely on static binding only, and they may occur only in static classes which cannot derive from interfaces or any other classes (see Sec. 2.3). Now, from a more functional point-of-view, the mechanism behind a Visitor can be seen as a multiple case function triggered by different object types. In other words, a concrete visitor can be represented as a functional value. A possible FP-oriented Visitor implementation is therefore to let *Accept* methods be of higher-order and take a concrete visitor as a functional argument. For instance, consider a classic implementation of a Visitor associated with a Composite over graphic figures (here, $\{C_i\} = \{$`SimpleFigure`, `CompositeFigure`$\}$):

```
public delegate T VisitorFun<V, T>(V f);

public interface IFigureF {
   String GetName();
   T Accept<T>(VisitorFun<IFigureF, T> v);
}

public class SimpleFigureF : IFigureF {
   private String _name;
   public SimpleFigureF(String name) { _name = name; }
   public String GetName() { return _name; }
   public T Accept<T>(VisitorFun<IFigureF, T> v) {
      return v(this);
   }
}
public class CompositeFigureF : IFigureF {
   private String _name;
   private IFigureF[] _figureArray;
   public CompositeFigureF(String name, IFigureF[] s) {
      _name = name; _figureArray = s;
   }
   public String GetName() { return name; }
   public T Accept<T>(VisitorFun<IFigureF, T> v) {
      foreach (IFigureF f in _figureArray) {
         f.Accept(v);
      }
      return v(this);
   }
}
```

Note here that a call to `Accept` yields a result: such a "*functional Visitor*" does not provide any method to its access internal states, if any, as it is usually the case for classic Visitors (see e.g. [OWG08]). A first solution to exploit this architecture consists of building methods of type `VisitorFun` which associate a function code with each visited figure type, dynamically checking type information. Such a constructor – corresponding to one concrete visitor $Visit_j$ – can be the following (here we again use the closure technique introduced in Sec 2.2):

```
public static VisitorFun<IFigureF, String>
   MakeNameFigureVisitorFun() {
      string _state = "";
      return obj => { if (obj is SimpleFigureF) _state += obj.GetName() + " "; else
                      if (obj is CompositeFigureF) _state += obj.GetName() + "/";
                      return _state; };
}
```

This solution shows similarities to *pattern matching filtering* over datatypes as in ML or Haskell (but without type safety). However, duplicated explicit type switching for function selection is not in the spirit of the Visitor pattern, and this solution is as hard to extend and adapt as the classic OOP implementation. A more sophisticated, but still simple, FP-oriented solution consists of using a *data-driven* (or *table-driven*) approach in order to obtain generic selection processes (see e.g. [ASS85]). In FP, this classic and easily implemented technique is based: (1) on building dictionaries of pairs $(key, function)$; (2) on implementing generic functions able to exploit these dictionaries. Accordingly, a FP-oriented Visitor using this technique is made of: (1) dictionaries of pairs in the form $(type, method)$, for which there is one entry for each visited class C_i (one dictionary corresponds to one concrete visitor $Visit_j$); (2) generic visitor methods able to exploit these dictionaries, that is, able to call the right method corresponding to a given type. In C#, such dictionaries can be implemented by using the predefined container type `Dictionary` and the dynamic typing capabilities of the language (`System.Type` is the predefined general type of types, `obj.GetType()` yields the type of an object `obj`, and `typeof(c)` yields the type of the objects of class `c`). Here is first a generic function which builds methods of type `VisitorFun` from specific dictionaries, and which associates the data-driven method selection process with them:

```
public delegate T VisitFun<V, T>(V unaryFun, T currentState);

public static VisitorFun<V, T>
  MakeVisitorFun<V, T>(Dictionary<System.Type, VisitFun<V, T>> Dict) {
     T state = default(T);
     return obj => { state = Dict[obj.GetType()](obj, state);
                     return state; };
}
```

Next, the following definitions correspond to the same functional concrete visitor example as above:

```
var dict1 = new Dictionary<System.Type, VisitFun<IFigureF, String>>();
dict1.Add(typeof(SimpleFigureF),    (f, s) => s + f.GetName() + " ");
dict1.Add(typeof(CompositeFigureF), (f, s) => s + f.GetName() + "/");

var nameFigureFunVisitor1 =  MakeVisitorFun<IFigureF, String>(dict1);
```

Extending such a Visitor with a new class satisfying the interface `IFigureF`, i.e. extending the set of the visited classes C_i, is not a problem any longer: one just has to extend the dictionaries Moreover, such dictionaries may evolve at runtime, and any method with the expected type may be inserted into them (cf. Sec. 4.2).

We have illustrated that using FP techniques may provide solutions to some of the drawbacks of the classic Visitor implementation, mainly, refactoring resistance, staticness and naming inflexibility. FP data-driven techniques indeed allow one to easily get ad-hoc selection processes, which can then be mixed with an OO architecture. However, these properties are often obtained at the expense of type safety (e.g., exceptions are raised if a method definition is not present) and possible type incoherences (e.g., combinations of unrelated methods, multiple definitions for the same case): the completeness and the validity of a given dictionary may remain a responsibility of the programmers.

6 CONCLUSION

In order to write powerful and flexible code at a function/method granularity design level within classic OOP, there exist essentially two ways:

1. *Single function/method encapsulation into objects*: this technique leads to homogeneous architecture design, only based on OOP concepts[9]. However, it also leads to syntactic overhead and architecture needless complexity, in particular by bloating the sets of classes and the inheritance graphs[10].
2. *Single function/method management by introspection/reflection and plug-in capabilities*: these techniques lead to sophisticated and ad-hoc solutions, but they also lead to insufficient safety and bug intractability, by being external to the language itself.

A possible answer to the inconveniences of these techniques is to include a typed first-class function/method granularity level, that is, to offer some FP capabilities to OOP. We have indeed seen that this paradigm combination provides some simplifications and increased flexibility in OOP in terms of:

- Code abstractions at a function/method level.
- Easy generic iterator/loop implementations.
- Easy operation compositions, sequence/query comprehensions.
- Function partial applications.
- Reductions of the number of object/class definitions.
- Name abstractions at a function/method level.

We have also seen that some characteristic FP techniques may enrich OOP with alternative solutions, in particular with respect to:

- Laziness emulations (e.g., in Virtual Proxies).
- Data-driven or table-driven programming (e.g., in Visitors).

However, we have stressed that the quality of these solutions could be impaired by problems mainly induced by the *design granularity mismatch*, particularly when classes are encapsulating units, like in Java, C++, or C# . Several aspects of these problems have been shown to be partially solved by specific modular organizations,

[9]For instance, in [JLT98], one reads "*[...] bound method references [delegates] are harmful because they detract from the simplicity of the Java programming language and the pervasively object-oriented character of the APIs. Bound method references also introduce irregularity into the language syntax and scoping rules.*".

[10]For instance, in [Bec97], one reads "*Managing a namespace across many classes is expensive. You would like to invoke the costs of a new class only when there is a reasonable payoff. A large family of classes with only a single method each is unlikely to be valuable*".

anonymous constructs, and interrelation mechanisms like *extension methods*. Still, homogeneity of one's architectural structure, reusability, extensibility, datatype coherence may remain difficult to deal with when some FP design level is mixed with OOP. As a conclusion, the claim at the beginning of the paper saying that "*adding some functional programming capabilities within an object-oriented language leads to benefits for object-oriented programming design*" is certainly true, but unsurprisingly without providing any silver bullet.

Acknowledgement: *I thank G. Eyrolles for many fruitful discussions.*

References

[Ale01] A. Alexandrescu. *Modern C++ Design. Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Bak93] Henry G. Baker. Iterators: signs of weakness in object-oriented languages. *OOPS Messenger*, 4(3):18–25, 1993.

[Bec97] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[BGGA08] G. Bracha, N. Gafter, J. Gosling, and P. Von der Ahé. Closures for the Java programming language (v0.5), 2008. (last seen in: http://www.javac.info/closures-v05.html).

[BGS07] K. Beck, E. Gamma, and D. Saff. Junit documentation – a cook's tour, July 2007.

[Bis08] J. M. Bishop. *C# 3.0 Design Patterns.* O'Reilly, 2008.

[Blo01] J. Bloch. *Effective Java (Programming Language Guide).* The Java Series. Addison-Wesley, 2001.

[Bru02] K. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics.* MIT Press, 2002.

[Bru03] K. Bruce. Some challenging typing issues in object-oriented languages. *Electronic notes in Theoretical Computer Science*, 82(8), 2003.

[BW88] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[Cli06] M. Cline. C++ FAQ Lite, 2006. (last seen in: http://www.parashift.com/c++-faq-lite).

[CMLC06] C. Clifton, T. Millstein, G.T. Leavens, and C. Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[Coc08] M. Cochran. Introduction to functional programming in C#. C# Corner, January 2008. (last seen in: http://www.c-sharpcorner.com).

[DHM+99] P. Dubois, M. Howard, B. Meyer, M. Schweitzer, and E. Stapf. From calls to agents. *Journal of Object-Oriented Programming*, 12(6), 1999.

[ECM06a] Standard ECMA. C# language specification, June 2006. Number-334.

[ECM06b] Standard ECMA. Eiffel: Analysis, design and programming language, June 2006. Number-367.

[FF98] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN Intl. Conference on Functional Prog. (ICFP '98)*, volume 34(1), pages 94–104, 1998.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[FWH92] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages.* MIT Press, 1992.

[Gaf07] N. Gafter. JSR proposal: Closures for Java, 2007. (last seen in: http://www.javac.info/consensus-closures-jsr.html).

[GHJV95] E. Gamma, R. Helm, R. Jonhnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Goe07] B. Goetz. Java theory and practice: The closures debate. Technical report, IBM, April 2007. (last seen in: http://www.ibm.com/developerworks/java/library/j-jtp04247.html).

[GR89] A. Goldberg and D. Robson. *Smalltalk-80 – the language.* Addison-Wesley, 1989.

[GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, 1991.

[HT06] Jay Hilyard and Stephen Teilhet. *C# Cookbook.* O'Reilly, 2006.

[Hud00] P. Hudak. *The Haskell School of Expression (Learning Functional Programming Through Multimedia).* Cambridge University Press, 2000.

[Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[JF09] J. Jarvi and J. Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 2009. (in press).

[JFC08] J. Jarvi, J. Freeman, and L. Crowl. Lambda expressions and closures: Wording for monomorphic lambdas. Technical Report N2550=08-0060, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2008.

[JLT98] The Java Language Team. About Microsoft's "delegates" (whitepaper). Technical report, JavaSoft, Sun Microsystems, Inc., 1998. (last seen in: http://java.sun.com/docs/white/delegates.html).

[Kee89] S. E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

[Ken06] A. Kennedy. C# is a functional programming language. Fun in Oxford Meeting, November 2006.

[KFF98] S. Krishnamurthi, M. Felleisen, and D.P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 91–113, London, UK, 1998. Springer-Verlag. LNCS 1445.

[Küh94] Th. Kühne. Higher order objects in pure object-oriented languages. *SIGPLAN Notices*, 29(7):15–20, 1994.

[Küh97] Th. Kühne. The function object pattern. *C++ Report*, 9(9):32–42, 1997.

[Küh99] Th. Kühne. *A Functional Pattern System for Object-Oriented Design.* Verlag Kovac, 1999.

[Lau95] K. Laufer. A framework for higher-order functions in C++. In *Proc. Conf. Object Oriented Technologies (COOTS)*, pages 103–116, June 1995.

[LDG+08] X. Leroy, Doligez D., J. Garrigue, Rémy D., and Vouillon J. *The Objective Caml System. Documentation and User's Manual.* INRIA, February 2008.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[Mar03] R. C. Martin. *Agile Software Development (Principles, Patterns and Practices).* Prentice Hall, 2003.

[Mei07] Erik Meijer. Confessions of a used programming language salesman. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 677–694, 2007.

[Met04] S.J. Metsker. *Design Patterns in C#.* Addison Wesley, 2004.

[Mey01] S. Meyers. *Effective STL.* Addison-Wesley, 2001.

[Mic98] Microsoft. The truth about delegates. Technical report, MSDN Newsletter, September 1998.

[Mic07a] Microsoft. C# Language Specification (Version 3.0), 2007.

[Mic07b] Microsoft. C# Programming Guide. Visual Studio 2008 Documentation, 2007.

[MS04] B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. Funct. Program.*, 14(4):429–472, 2004.

[Nar05] Ph. Narbel. *Programmation fonctionnelle, générique et objet (Une introduction avec le langage OCaml).* Vuibert, Paris, 2005.

[Nar07] Ph. Narbel. A multiparadigmatic study of the object-oriented design patterns. In *MPOOL'07, European Conference on Object-Oriented Programming*, Berlin, July 2007.

[Ode08] M. Odersky. *The Scala Language Specification.* Programming Methods Laboratory, EPFL, 2008.

[Ost08] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.

[OWG08] B. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08: Proc. of the 23nd conference on Object oriented programming systems and applications*, pages 439–456, 2008.

[Pau97] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1997.

[Pet07] Th. Petricek. Lazy computation in C#. Technical report, Microsoft, Visual C# Developer Center, October 2007.

[PJ98] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Computer Software and Applications Conf., 1998. COMPSAC '98*, pages 9–15, 1998.

[Rea89] Ch. Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[Rey75] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages*, pages 157–168. INRIA, IFIP Working Group 2.1 on Algol, 1975. Schuman, A. (Editor).

[Ric01] Jeffrey Richter. An introduction to delegates. MSDN Magazine, April 2001.

[SF90] G. Springer and D.P. Friedman. *Scheme and the Art of Programming.* MIT Press, 1990.

[SL95] A. A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard, Palo Alto, CA, 1995.

[SM00] Y. Smaragdakis and B. McNamara. Bridging functional and object-oriented programming. Technical Report 00-37, Georgia Tech CoC Tech., 2000.

[VMR98] J. Vlissides, R. Martin, and D Riehle. *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[Wad98] Ph. Wadler. The expression problem. Java Genericity Mailing List, 1998.

[Wie08] Richard Wiener. Arithmetic function interpreter in C# 3.0 using lambda expression trees. *Journal of Object Technology*, 7(3):41–48, 2008.

ABOUT THE AUTHOR

Ph. Narbel is associate professor in computer science at Bordeaux University, France. He can be reached at `narbel_at_labri.fr`.