Functional Programming at Work in Object-Oriented Programming

Narbel

version 2010
A Claim about Programming Styles

**Claim:**

Adding functional programming capabilities to an object-oriented language leads to benefits in object-oriented programming design.
Some old and less old languages with FP+OOP:

- For instance, **Smalltalk**, **Common Lisp** (CLOS).
- More recently, **Python** or **Ruby**.

*Notations:* **FP**, Functional programming; **OOP**, Object-oriented programming,
Practices in OOP languages include **emulations of FP techniques**:

- C++ programmers: *function pointers and overloading of the () operator*, i.e. “object-functions” or functors.

- Java programmers: *anonymous classes* and introspection/reflexion.
The idea of **using FP to enrich OOP** is old, see e.g. the discussions about the **problem of the user-defined datatype extension**:

- *User-defined types and procedural data structures as complementary approaches to data abstraction*. Reynolds. 1975.

A recent trend: to propose and include typed **FP extensions in mainstream static OO languages**.

- Extensions for **C++** (see e.g. Laufer, Striegnitz, McNamara, Smaragdakis), and work in progress in the C++ standard committees.
- **Java 7** expected to include FP constructs.
- **C#** offers FP constructs (even more in its 3.0 version).

- Also in modern research languages with sophisticated typed frameworks: e.g., **OCaml, Scala**.
Purpose of the talk

- **Mix of FP with OOP** not so much practiced.

⇒ **Purpose of this talk**: a practical synthesis about what a programmer can expect when FP is available in OOP (using C# 3.0).
Some References


- **Synthesizing Object-Oriented and Functional Design to Promote Re-Use.** Krishnamurthi, Felleisen, Friedman. 1998.


- **Bridging Functional and Object-Oriented Programming (FC++)**. Smaragdakis, McNamara. 2000.

- **Confessions of a used programming language salesman.** Meijer. OOPSLA’07.

- **C# 3.0 Design Patterns.** J. M. Bishop. O’Reilly, 2008.
Main Points

Specific points discussed in this talk:

1. Criteria to check that FP techniques are possible.
2. Idiomatic and architectural effects of having FP capabilities in OOP.
3. FP analysis of some classic OO design patterns.
Criteria for FP techniques

Criterion:
First-Class Values. Functions/methods should be first-class citizens.

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

Remark: First-class functions generally ⇒ anonymous functions.
Criterion:

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

Rule:

*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.*
Granularity Mismatch

- FP and OOP operate on **different design granularity levels**:
  - **Functions/methods**: “programming in the small” level.
  - **Classes/objects/modules**: “programming in the large” level,

⇒ **At least two questions**:
  - *Where do we locate the source of individual functions in an OOP architecture?*
  - *How do we relate such individual functions to an OOP architecture?*

⇒ **design granularity mismatch**.
Criteria for FP techniques

Criterion:

**FP-OOP interrelation tools**: Standalone functions/methods should be explicitly relatable to the class/object level.

Rule:

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

Criterion:

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

Rule:

*When Criterion 4 holds, an OOP language acknowledges the fact that FP is one of its fully integrated tool.*
C# offers a FP feature called **delegates**:

```csharp
delegate string StringFunType(string s); // declaration

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

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

Delegates are **first-class values**.
Delegate types *can type method parameters*, and delegates *can be passed as arguments* as any other values:

```csharp
string Gf1(StringFun f, string s) {
    // delegate f as a parameter
    // call
    WriteLine(Gf1(G1, "some string");
```
2. Delegates as First-Class Values

Delegates can be returned as a computation of a method. For instance, assuming \( G \) is a method of type \( \text{string} \Rightarrow \text{string} \) and implemented in \( \text{SomeClass} \):

```csharp
class StringFun

Gf2() { //delegate as a return value
    ...
    return (new SomeClass()).G;
}

WriteLine(Gf2()("some string")); //call
```
Delegates as First-Class Values

Delegates can take place into data structures:

```csharp
var l = new LinkedList&lt;StringFun&gt;();
    // list of delegates

[...]
l.AddFirst(G1); // insertion of a delegate in the list
WriteLine(l.First.Value("some string"));
    // extract and call
```
Anonymous delegates

- C# delegates may be **anonymous**:

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

- Anonymous delegates can look even more like **lambda expressions**:

  ```csharp
  (s => { return s + "some string" });
  s => s + "some string";
  ```
No strict closures in C# (intrinsically “impure FP”):

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

⇒ Usual enclosing technique in impure FP:

```
StringFun F() {
    int _counter = 1000;
    return s => { return s + _counter.ToString(); };
}
```
**Extension methods**: enable a programmer to add methods to existing classes without creating new derived classes:

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

String s1 = "aaa bb cccc";

String s1 = "some chain";

s1.SimpleWordCount(); // usable as a String method
SimpleWordCount(s1); // also usable as a standalone method
Another **classic example** of extension method:

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

List<int> someList = [...];
someList.MySort();
```
An Interrelation FP/OOP: the Extension Methods

- Functions/methods implemented for delegates are often defined as extension methods.

- Extension methods: related to "open-classes" (see e.g., CLOS, Ruby, Multijava).

- Extension methods have harsh constraints in C#:
  - Only static!
  - Not polymorphic (not virtual)!

- For Java 7, closure conversions are proposed.

- In Groovy, explicit closure conversions exist.
**Basic Delegates Predefinitions.** C# offers functional and procedural generic delegate predefined types for arity up to 4... (respectively under the name `Func` and `Action`): 

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

...[...]

**NB:** overloading applies for generic delegates too.
First-Class Multiple Invocation and Multicasting. A delegate may itself contain an “invocation list” of delegates.

When such delegate is called, methods of the included delegate are invoked in the order in which they appear in the list.

The result value is determined by the last method called in the list.

Management of multicasting: + and – are overloaded to act on these invocation lists:

```csharp
menuItem1.Click += [...]; // some delegate
```
Function Marshalling and Serialization. C# allows lambda expressions to be represented as data structures called *expression trees*:

```
Expression<Func<int, int>> expression = x => x + 1;

var d = expression.Compile();
d.Invoke(2);
```

As such, they may be stored and transmitted.
Some **idiomatic and technical effects** of having FP capabilities in OOP:

1. Code factoring at a function/method granularity level,
2. Generic iterator and loop operations
3. Operation compositions (and sequence comprehensions).
4. Function partial applications and currying.
A simple code:

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

With **functional abstraction**:

```c++
public delegate int Fun(int x, int y, int z);
float MFun(Fun f, int x2, int y) {
    int x1 = [...] ;
    [...]  
    f(x1, x2, y);
    [...]  
}
```

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

⇒ **No local duplications + separation of concerns.**
A simple and effective application of the functional abstraction: generic **higher-order iterated operations over data**.

For instance, the **internal iterators** (Maps):

```csharp
IEnumerable<T2>
Map<T1, T2>(this IEnumerable<T1> data, Func<T1, T2> f) {
    foreach (T1 x in data)
        yield return f(x);
}

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

- **FP ⇒ Easy operation compositions.**

- An initial method code:

```csharp
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)
        Console.WriteLine("{0}: {1}", x.Key, x.Value.ToString());
}
```
A first factoring using higher-order functions:

```csharp
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 => Console.WriteLine("{0}: {1}",
                                 x.Key, x.Value.ToString()));
}
```
A second factoring using extension methods:

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

⇒ Increased readability.
Operation Compositions

In C#, such operation compositions are often used with the “Language Integrated Query” (LINQ) – defined to unify programming with relational data or XML, e.g. (Meijer):

```csharp
var q = programmers
    .Where(p => p.Age > 20)
    .OrderByDescending(p => p.Age)
    .Select(g => new{
        Language = g.Key,
        Size = g.Count(),
        Names = g
    });
```

⇒ Solve some of the “impedance mismatch” between OOP and data base exploitation.
With first-class functions, every \( n \)-ary function can be transformed into a composition of \( n \) unary functions, that is, into a \textbf{curried function}:

\[
\text{Func}\langle \text{int}, \text{int}, \text{int} \rangle \text{ lam1} = (x, y) \Rightarrow x + y;
\]
\[
\text{Func}\langle \text{int}, \text{Func}\langle \text{int}, \text{int} \rangle \rangle \text{ lam2} = x \Rightarrow (y \Rightarrow x + y);
\]
\[
\text{Func}\langle \text{int}, \text{int} \rangle \text{ lam3} = \text{lam2}(3); \quad \text{// partial application}
\]

\textbf{Curryfying:}

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

\[
\text{Func}\langle \text{int}, \text{int} \rangle \text{ lam4} = \text{lam1}.\text{Curry}()(3); \quad \text{// partial application}
\]
Some architectural effects of having FP capabilities in OOP:

1. Reduction of the number of object/class definitions.
2. Name abstraction at a function/method level.
3. Operation compositions (and sequence comprehensions).
4. Function partial applications and currying.
Limitation of the Number of Object/Class Definitions

- Functional abstraction ⇒ Avoid cluttering the OO architecture with new classes:

```java
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 M(IFun funobj, int x2, int y) {
    int x1 = [...val1...]
    [...]
    funobj.F(x1, x2, y);
    [...]
}

int z1 = M(new F1(), 1, 2);
int z2 = M(new F2(), 3, 4);
```
Using first-class methods allows parameters to be instantiated by any method satisfying their declared types.

```java
interface IStringFun {
  string F1(string s);
}

IStringFun obj1 = [...] ;
[...] obj1.F1 [...] 
```

⇒ **Name abstraction.**

⇒ **Induce some “structuralness” into nominal-oriented OOP, i.e. flexibility.**
Example of *name abstraction* with a *Bridge*:

- The initial code:
  ```java
  public class Window {
      private WindowSys _imp;
      void DrawFigure ([...]) {
          _imp.DeviceFigure ([...]);
      }
  }
  ```

- With delegates:
  ```java
  public delegate void DeviceFigureFun ([...]);
  public class Window {
      private DeviceFigureFun _devfig;
      void DrawFigure ([...]) {
          _devfig ([...]);
      }
  }
  ```

- Without delegates: *must use Adapters* (see e.g. `ActionListener` in Java).
Another example of *name abstraction* with an Abstract Factory:

The initial code:

```java
public interface Maze {
    // ...
}

public interface Wall {
    // ...
}

public interface Room {
    // ...
}

public interface MazeFactory {
    Maze MakeMaze();
    Wall MakeWall();
    Room MakeRoom();
}
```

With delegates:

```java
public delegate Maze MakeMazeFun();

public delegate Wall MakeWallFun();

public delegate Room MakeRoomFun();

public abstract class MazeFactoryFun {
    MakeMazeFun MakeMaze;
    MakeWallFun MakeWall;
    MakeRoomFun MakeRoom;
}
```
Name Abstraction at a Method Level

- **Name abstraction** ⇒ More flexibility.

- But also ⇒ More looseness and problems of architecture organization.
FP Design Granularity Mismatch

- Where do we put the sources of the standalone methods?:

  - **Solution with Basic Modules**: functions as static methods in some utility class or module.
    ⇒ **Not easily extensible, and does not mix so well with class hierarchy.**

  - **Solution with anonymous constructs**: function implementations of function directly into the calls:
    
    ```
    int z1 = MFun( (x1, x2, x3) => [...] <F1 code>... ], 1, 2);
    int z2 = MFun( (x1, x2, x3) => [...] <F2 code>... ], 3, 4);
    
    ⇒ **Spread the code all over the calls, and may lead to no reusability.**
    ```
Some of the **classic OOP Design Patterns** can be considered under FP influence:...

2. Observer.
3. Proxy.
4. Visitor.
A **Strategy** pattern is to *let an algorithm vary independently of clients that use it.*

(The pattern figures are taken from the GoF book.)
A **Strategy**: just a case of abstracting code at a method level (⇒ No need of OO encapsulation and new class hierarchies).

For instance in the .NET Framework:

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

public delegate bool Predicate<T>(T obj)
public List<T> FindAll(Predicate<T> match)
```
Strategy with FP
The **Command** pattern *encapsulates requests (method calls) as objects* so that they can easily be transmitted, stored, and applied.

⇒ **Same as Strategy.**

For instance, *menu implementations*:

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

private void menuItem1_Click(object sender, System.EventArgs e) {
    OpenFileDialog fd = new OpenFileDialog();
    fd.DefaultExt = ".*"; fd.ShowDialog();
}

public void CreateMyMenu() {
    MainMenu mainMenu1 = new MainMenu();
    MenuItem menuItem1 = new MenuItem();
    [...]
    menuItem1.Click += new System.EventHandler(menuItem1_Click);
}
```
Command with FP

Diagram:

- Application
  - Add(Document)
- Menu
  - Add(MenuItem)
- MenuItem
  - Clicked()
  - command
  - command->Execute()
- Command
  - Execute()
The **Observer** pattern: *a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated.*
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);
        }
    }
}
FP ⇒ Observer with functional values as updaters:

```csharp
public delegate void UpdateFun<T>(T s);

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

    public void Attach(UpdateFun<T> f) {
        _updateHandler += f;
    }

    public void Notify(T s) {
        _updateHandler(s);
    }
}
```

⇒ No need of observer classes with methods called Update().
The **Virtual Proxy** pattern: *placeholders for other objects such that their data are created/computed only when needed.*
public class SimpleProxy : I {
    private Simple _simple;
    private int _arg;
    protected Simple GetSimple() {
        if (_simple == null)
            this._simple = new Simple(this._arg);
        return _simple;
    }
    public SimpleProxy(int i) { this._arg = i; }
    public void Process() {
        GetSimple().Process();
    }
}
Virtual Proxy with FP Lazyness

- FP ⇒ **Less need of specific Proxy classes:**

```java
public class SimpleLazyProxy : I {
    private Lazy<Simple> _simpleLazy;
    public SimpleLazyProxy(int i) {
        this._simpleLazy = new Lazy<Simple>(() => new Simple(i));
    }
    public void Process() {
        this._simpleLazy.Value.Process();
    }
}
```
The **Visitor** pattern: *lets you define new operations without changing the classes of the elements on which they operate*

Without Visitors, each subclass of a hierarchy has to be edited or derived separately.

NB: Visitors are **at the crux of many of the programming design problems**...
public interface IFigure {
  String GetName();
  void Accept<T>(IFigureVisitor<T> v);
}

public class SimpleFigure : IFigure {
  private String _name;
  public SimpleFigure(String name) { this._name = name; }
  public String GetName() { return this._name; }
  public void Accept<T>(IFigureVisitor<T> v) {
    v.Visit(this);
  }
}

public class CompositeFigure : IFigure {
  private String _name;
  private IFigure[] _figureArray;
  public CompositeFigure(String name, IFigure[] s) {
    this._name = name; this._figureArray = s;
  }
  public String GetName() { return this._name; }
  public void Accept<T>(IFigureVisitor<T> v) {
    foreach (IFigure f in _figureArray)
      f.Accept(v);
    v.Visit(this);
  }
}
public interface IFigureVisitor<T> {
    T GetVisitorState();
    void Visit(SimpleFigure f);
    void Visit(CompositeFigure f);
}

public class NameFigureVisitor : IFigureVisitor<string> {
    private string _fullName = "";
    public string GetVisitorState() { return _fullName; }
    public void Visit(SimpleFigure f) {
        _fullName += f.GetName() + " ";
    }
    public void Visit(CompositeFigure f) {
        _fullName += f.GetName() + " /";
    }
}
Weaknesses of Visitors

- Some well-known weaknesses of Visitors:
  - **Refactoring Resistance.** A Visitor definition is dependent on the set of client classes on which it operates.
  - **Staticness.** A Visitor is static in its implementation (type-safety but less flexibility).
  - **Invasiveness.** A Visitor needs that the client classes anticipate and/or participate in making the selection of the right method.
  - **Naming Inflexibility.** A Visitor needs that all the different implementations of the visit methods be similarly named.
Visitor and Extension Methods

- An attempt to solve Visitor problems with extension methods (cf. “open classes” – but not ok in C#):

```csharp
public interface IFigure {
    String GetName();  // no Accept method required
}

[...]

public static class NameFigureVisitor {
    public static void NameVisit(this SimpleEFigure f) {
        _state = f.GetName() + " " + _state;
    }

    static void NameVisit(this CompositeFigure f) {
        _fullName = f.GetName() + ":" + _fullName;
        foreach (IFigure g in f.GetFigureArray())
            g.NameVisit();  // !!! dynamic dispatch required...
    }

    [...]
}
```

Narbel
Functional Programming at Work in Object-Oriented Programming
Visitor with FP

FP ⇒ Visitors can be functions:

```java
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) { this._name = name; }
    public String GetName() { return this._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) {
    this._name = name; this._figureArray = s;
}

public String GetName() { return this._name; }

public T Accept<T>(VisitorFun<IFigureF, T> v) {
    foreach (IFigureF f in _figureArray) {
        f.Accept(v);
    }

    return v(this);
}
}
Visitor with FP
A simple functional Visitor:

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

But ⇒ Ad-hoc explicit selection needed...
A **Visitor** with **functional data-driven programming** made of:

- Dictionaries of pairs in the form \((\text{type}, \text{method})\).
- Generic “accept” able to exploit these dictionaries and call the right method corresponding to a given type.

⇒ **Explicit generic selection mechanism.**
Use of data-driven oriented Visitor:

```csharp
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);
```
Visitor with Functional Data-Driven Programming

- FP ⇒ Data-driven Functional Visitors.
- ⇒ Less refactoring resistance, less name rigidity, and less staticness.
- But ⇒
  - Possible type incoherences...
  - Syntax intricacies...
In order to get flexible code in classic OOP at a method level, essentially two ways:
In order to get flexible code in classic OOP at a method level, essentially two ways:

- **Method encapsulation in objects** (rich but heavy).
In order **to get flexible code in classic OOP at a method level**, essentially two ways:

- **Method encapsulation in objects** (rich but heavy).
- **Method management by introspection/reflection and plug-in capabilities** (flexible but type unsafe and technically cumbersome).
In order to get flexible code in classic OOP at a method level, essentially two ways:

- **Method encapsulation in objects** (rich but heavy).
- **Method management by introspection/reflection and plug-in capabilities** (flexible but type unsafe and technically cumbersome).

⇒ A possible answer is to include a typed first-class method granularity level.
Summing Up

- **OOP with FP granularity level** ⇒
  - Code Abstraction at a function/method level.
  - Convenient generic iterator/loop implementations.
  - Operation compositions, sequence/query comprehensions.
  - Function partial applications.
  - Limitations of the number of object/class definitions.
  - Name abstractions at a function/method level.

- And:
  - Lazyness emulations (used e.g. in Virtual Proxies).
  - Data-driven or table-driven programming (used e.g. in Visitors).
Summing up

- **FP + OOP ⇒**
  - Architecture simplifications.
  - Increased flexibility.
Summing up

- **FP + OOP** ⇒
  - Architecture simplifications.
  - Increased flexibility.

- But ⇒ **Design granularity mismatch** (functions at a finer design level than classes/objects/modules) ⇒
  - Architecture inhomogeneity.
  - Lack of type coherence.
  - No easy reusability.
Summing up

- **FP + OOP** ⇒
  - Architecture simplifications.
  - Increased flexibility.

- But ⇒ **Design granularity mismatch** (functions at a finer design level than classes/objects/modules) ⇒
  - Architecture inhomogeneity.
  - Lack of type coherence.
  - no easy reusability.

- Some **partial solutions** of granularity mismatch:
  - Specific modular organizations.
  - Anonymous constructs.
  - Interrelation means like “extension methods”.
The **Claim**: Adding functional programming capabilities to an object-oriented language leads to benefits in object-oriented programming design.
The Claim:

Adding functional programming capabilities to an object-oriented language leads to benefits in object-oriented programming design.

OK, but without being a silver bullet...
The **Claim**: 

*Adding functional programming capabilities to an object-oriented language leads to benefits in object-oriented programming design.*

**OK**, but **without being a silver bullet**...

Remark: anyway, FP is expected in Java, C++, Sprutch...