Draft F# Component Design Guidelines (August 2010)

This documentation is a draft set of component design guidelines for F# programming related to the 2.0 release of F# made by Microsoft Research and the Microsoft Developer Division in April 2010.

This document assumes you are familiar with F# programming.

For more information on F# programming see fsharp.net. The F# team is always very grateful for feedback on these guidelines. You can submit feedback by emailing fsbugs@microsoft.com. Many thanks to the F# user community for their helpful feedback on the document so far.

1 Overview

This document looks at some of the issues related to F# component design and coding. In particular, it covers:

- Guidelines for designing "vanilla" .NET libraries for use from any .NET language.
- Guidelines for F#-to-F# libraries and F# implementation code.
- Suggestions on coding conventions for F# implementation code.

F# is often seen as a functional language, but in reality is a multi-paradigm language; the OO, functional and imperative paradigms are all well supported. That is, F# is a *functional-oriented* language—many of the defaults are set up to encourage functional programming, but programming in the other paradigms is effective and efficient, and a combination is often best of all. It is a common misconception that the functional and object-oriented programming methodologies are competing. In fact, they are generally orthogonal and largely complementary. Often, functional programming plays a stronger role "in the small" (e.g. at the implementation level of functions/method and the code contained therein) and OO plays a bigger role "in the large" (e.g. at the structural level of classes, interfaces, and namespaces, and the organization of APIs for frameworks).

Regardless of the methodology, the component and library designer faces a number of practical and prosaic issues when trying to craft an API that is most easily usable by developers. One of the strengths of the .NET platform is its *unified programming model* that is independent of the programming language being used. The consistency throughout both the .NET Framework and other .NET libraries is the result of conscientious application of the *.NET Library Design Guidelines*, published online by Microsoft and as a book ("Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries" by Krzysztof Cwalina and Brad Abrams) by Addison-Wesley. These guidelines steer library designers towards creating a consistent set of APIs which enables components to be both easily authored in, and seamlessly consumed by, a variety of .NET languages.

As a .NET programming language, the general guidelines and conventions for .NET component programming and library design apply to F#. Nevertheless F# has a number of unique features, as well as some of its own conventions and idioms, which make it worthwhile to provide prescriptive advice specific to using F#. Even if you are writing small F# scripts, it can be useful to be familiar with these design guidelines, as today's scripts and tiny projects often evolve into tomorrow's reusable library components.

The .NET Library Design Guidelines are described in terms of conventions and guidelines for the use of the following constructs in public framework libraries:

- Assemblies, namespaces, and types
- Classes and interfaces, containing properties, methods, and events
- .NET delegate types
- Enumerations (that is, enums from languages such as C#)

- Constants (that is, constant literals from languages such as C#)
- Type parameters (that is, generic parameters)

From the perspective of F# programming, you must also consider a variety of other constructs, including:

- Union types and record types
- Values and functions declared using `let` and `let rec`
- Modules
- Function types
- F# optional parameters and extension methods

Good framework library design is always nontrivial and often underestimated. F# framework and library design methodology is inevitably strongly rooted in the context of .NET object-oriented programming. In this document, we give our guidelines on how you can go about approaching library design in the context of F# programming. As with the .NET Library Design Guidelines, these guidelines are not completely proscriptive –ultimately the final choices lie with F# programmers and software architects.

A primary decision point is whether you are designing components for use exclusively from F#, or whether your components are intended for use from other .NET languages (like C# and Visual Basic). APIs aimed specifically at F# developers can and should take advantage of the variety of F#-specific types and constructs that provide some of the unique strengths of the F# language. While it is possible to consume all of these components from other languages, a well-designed API designed for developers of any .NET language should use a more restricted subset of F# in the public interface, so as to provide familiar, idiomatic APIs that are consistent with the rest of .NET Framework.

Sections 2-4 give recommendations for authoring F# libraries depending on the library's intended audience. First in Section 2 we provide universal guidelines for all F# libraries. Next in Section 3 we offer advice regarding APIs designed specifically for F# developers. Then in Section 4 we describe recommendations for libraries that are intended to be consumed by any .NET language. These sections together provide all our advice regarding the design of the public interface published by an F# assembly.

Section 5 offers suggestions for F# coding conventions. Whereas the other sections focus on the component design, this section suggests some recommendations regarding style and conventions for general F# coding (e.g. the implementation of internal components).

The document concludes with an appendix containing an extended design example.

2 General Guidelines

There are a few universal guidelines that apply to F# libraries, regardless of the intended audience for the library. For all F# libraries, we propose the guidelines below. The first bullet is paramount, echoing one of the main themes of this document.

☑ **Do** be familiar with the .NET Library Design Guidelines

Regardless of the kind of F# coding you are doing, it is valuable to have a working knowledge of the .NET Library Design Guidelines. Most other F# and .NET programmers will be familiar with these guidelines, and expect .NET code to conform to them.

This in turn may require familiarity with C# and/or Visual Basic coding techniques.

The .NET Library Design Guidelines provide lots of general guidance regarding naming, designing classes and interfaces, member design (properties, methods, events, …) and more, and are a useful first point of reference for a variety of design guidance.

☑ **Do** add XML documentation comments to your code.

XML documents on public APIs ensure that users can get great Intellisense and Quickinfo when using these types and members, and enable building documentation files for the library. See the F# documentation about various xml tags that can be used for additional markup within xmldoc comments.

✓
```
/// A class for representing (x,y) coordinates
type Point =
    /// Computes the distance between this point and another
    member DistanceTo : anotherPoint:Point -> float
```

☑ **Consider** using explicit signature files (.fsi) for stable library and component APIs.

Using explicit signatures files in an F# library provides a succinct summary of public API, which both helps to ensure that you know the full public surface of your library, as well as provides a clean separation between public documentation and internal implementation details. Note that signature files add friction to changing the public API, by requiring changes to be made in both the implementation and signature files. As a result, signature files should typically only be introduced when an API has become solidified and is no longer expected to change significantly.

3 Guidelines for F#-Facing Libraries

In this section, we will present recommendations for developing public F#-facing libraries, that is, libraries exposing public APIs that are intended to be consumed by F# developers. (For guidance for internal/private F# implementation code, see the Section 5.) There are a variety of library-design recommendations applicable specifically to F#. In the absence of specific recommendations below, the .NET Library Design Guidelines are the fallback guidance.

3.1 Naming Conventions

☑ **Do** use the .NET naming and capitalization conventions for object-oriented code, including F#-facing libraries.

Table 2. Conventions Associated with Public Constructs in .NET Frameworks and Extensions for F# Constructs in F#-to-F# libraries

Construct	Case	Part	Examples	Notes
Concrete types	PascalCase	Noun/ adjective	`List`, `Double, Complex`	Concrete types are structs, classes, enumerations, delegates, records, and unions. Though type names are traditionally lowercase in OCaml, F# has adopted the .NET naming scheme for types.
DLLs	PascalCase		`Fabrikom.Core.dll`	
Union tags	PascalCase	Noun	`Some`, `Add`, `Success`	Do not use a prefix in public APIs. Optionally use a prefix when internal, such as `type Teams = TAlpha | TBeta | TDelta`.
Event	PascalCase	Verb	`ValueChanged`	
Exceptions	PascalCase		`WebException`	Name should end with "Exception".
Field	PascalCase	Noun	`CurrentName`	
Interface types	PascalCase	Noun/ adjective	`IDisposable`	Name should start with "I".
Method	PascalCase	Verb	`ToString`	
Namespace	PascalCase		`Microsoft.FSharp.Core`	Generally use `<Organization>.<Technology>[.<Subnamespace>]`, though drop the organization if the technology is independent of organization.
Parameters	camelCase	Noun	`typeName`, `transform`, `range`	
`let` values (internal)	camelCase	Noun/ verb	`getValue`, `myTable`	
`let` values (external)	camelCase …or… PascalCase	Noun	`List.map`, `Dates.Today`	`let`-bound values are often public when following traditional functional design patterns. However, generally use PascalCase when the identifier can be used from other .NET languages.
Property	PascalCase	Noun/ adjective	`IsEndOfFile`, `BackColor`	Boolean properties generally use `Is` and `Can` and should be affirmative, as in `IsEndOfFile`, not `IsNotEndOfFile`.

Table 2 summarizes the .NET guidelines for naming and capitalization in code. We have added our own recommendations for how these should be adjusted for some F# constructs.

Be aware of the following specific guidelines:

- The .NET guidelines discourage the use of abbreviations (for example, "use `OnButtonClick` rather than `OnBtnClick`"). Very common abbreviations, such as "Async" for "Asynchronous", are tolerated. This guideline is sometimes ignored for functional programming; for example, `List.iter` uses an abbreviation for "iterate". For this reason, using abbreviations tends to be tolerated to a greater degree in F#-to-F# programming, but should still generally be avoided in public component design.
- Acronyms such as XML are not abbreviations and are widely used in .NET libraries in uncapitalized form (`Xml`). Only well-known, widely recognized acronyms should be used.
- The .NET guidelines say that casing alone cannot be used to avoid name collisions, since some client languages (e.g. Visual Basic) are case-insensitive.

☑ **Do** use PascalCase for generic parameter names in public APIs, including for F#-facing libraries. In particular, use names like `T`, `U`, `T1`, `T2` for arbitrary generic parameters, and when specific names make sense, then for F#-facing libraries use names like `Key`, `Value`, `Arg` (but not e.g. `TKey`).

☑ **Do** use either PascalCase or camelCase for public functions and values in F# modules. camelCase is generally used for public functions which are designed to be used unqualified (e.g. `invalidArg`), and for the "standard collection functions" (e.g. `List.map`). In both these cases, the function names act much like keywords in the language.

3.2 Object, Type and Module Design

☑ **Do** use namespaces or modules to contain your types and modules.
Each F# file in a component should begin with either a namespace declaration or a module declaration.

✓ `namespace Fabrikom.BasicOperationsAndTypes`

```
type ObjectType1() = ...

type ObjectType2() = ...

module CommonOperations =
    ....
```

or

✓ `module Fabrikom.BasicOperationsAndTypes`

```
type ObjectType1() = ...
type ObjectType2() = ...


module CommonOperations =
    ....
```

From the F# coding perspective there is not a lot of difference between these options: the use of a module allows utility functions to be defined as part of the module, and is useful for prototyping, but these should be made private in a public-facing component. The choice affects the compiled form of the code, and thus will affect the view from other .NET language.

☑ **<u>Do</u>** use properties and methods for operations intrinsic to types.

This is called out specifically because some people from a functional programming background avoid the use of object oriented programming together, preferring a module containing a set of functions defining the intrinsic functions related to a type (e.g. `length foo` rather than `foo.Length`). But see also the next bullet. In general, in F#, the use of object-oriented programming is preferred as a software engineering device. This strategy also provides some tooling benefits such as Visual Studio's "Intellisense" feature to discover the methods on a type by "dotting into" an object.

For example:

```
✓ type HardwareDevice with
      ...
      member this.ID: string
      member this.SupportedProtocols: seq<Protocol>

✓ type HashTable<'Key,'Value> with
      ...
      new: IEqualityComparer<'Key> -> HashTable<'Key,'Value>
      member this.Add           : 'Key * 'Value -> unit
      member this.ContainsKey   : 'Key -> bool
      member this.ContainsValue : 'Value -> bool
```

☑ **<u>Do</u>** use classes to encapsulate mutable state, according to standard OO methodology.

In F#, this only needs to be done where that state is not already encapsulated by another language construct, e.g. a closure, sequence expression, or asynchronous computation.

For example:

```
✓ type Counter() =
      let mutable count = 0
      member this.Next() =
          count <- count + 1
          count
```

☑ **<u>Do</u>** use interface types to represent related groups of operations that may be implemented in multiple ways.

In F# there are a number of ways to represent a dictionary of operations, such as using tuples of functions or records of functions. In general, we recommend you use interface types for this purpose.

```
✓ type ICounter =
      abstract Increment : unit -> unit
      abstract Decrement : unit -> unit
      abstract Value : int
```

In preference to:

```
× type CounterOps =
      { Increment : unit -> unit
        Decrement : unit -> unit
        GetValue : unit -> int }
```

☑ **Consider** using the "module of collection functions" pattern (e.g. standard set of operations like `CollectionType.map` and `CollectionType.iter`) for new collection types.

```
module CollectionType =

    let map f c = ...

    let iter f c = ...
```

If you include such a module, follow the standard naming conventions for functions found in `FSharp.Core.dll`.

☑ **Consider** using the "module of top-level functions" design pattern for common, canonical functions, especially in math and DSL libraries.

For example, `Microsoft.FSharp.Core.Operators` is an automatically opened collection of top-level functions (like `abs` and `sin`) provided by FSharp.Core.dll.

Likewise, a statistics library might include a module with functions `erf` and `erfc`, where this module is designed to be explicitly or automatically opened.

☑ **Consider** using the `[<RequiredQualifiedAccess>]` and `[<AutoOpen>]` attributes if this improves the default ease of use and long-term maintainability of the library in common situations.

Adding the `[<AutoOpen>]` attribute to a module means the module will be opened when the containing namespace is opened. The `[<AutoOpen>]` attribute may also be applied to an assembly indicate a namespace or module that is automatically opened when the assembly is referenced.

For example, a statistics library `MathsHeaven.Statistics.dll` might contain a module `MathsHeaven.Statistics.Operators` containing functions `erf` and `erfc`. It is reasonable to mark this module as `[<AutoOpen>]`. This means "`open MathsHeaven.Statistics`" will also open this module and bring the names `erf` and `erfc` into scope. Another good use of `[<AutoOpen>]` is for modules containing extension methods.

Overuse of `[<AutoOpen>]` leads to polluted namespaces, and the attribute should be used with care. For specific libraries in specific domains, judicious use of `[<AutoOpen>]` can lead to better usability.

Adding the `[<RequireQualifiedAccess>]` attribute to a module indicates that the module may not be opened and that references to the elements of the module require explicit qualified access. For example, the Microsoft.FSharp.Collections.List module has this attribute.

This is useful when functions and values in the module have names that are likely to conflict with names in other modules and requiring qualified access can greatly increase the long-term maintainability and evolvability of a library: functions can be added to the module without breaking source compatibility.

☑ **Consider** defining operator members on types where using well-known operators is appropriate.

For example

```
✓ type Vector(x:float) =

    member v.X = x

    static member (*) (vector:Vector, scalar:float) =
        Vector(vector.X * scalar)

    static member (+) (vector1:Vector, vector2:Vector) =
        Vector(vector1.X + vector2.X)

  let v = Vector(5.0)
  let u = v * 10.0
```

This guidance corresponds to general .NET guidance for these types. However, it can be additionally important in F# coding as this will allow these types to be used in conjunction with F# functions and methods with member constraints, such as `List.sumBy`.

☑ **Consider** using method overloading for member functions, if doing so provides a simpler API.

☑ **Do** hide the representations of record and union types if the design of these types is likely to evolve.

The rationale for this is to avoid revealing concrete representations of objects. For example, the concrete representation of `System.DateTime` values is not revealed by the external, public API of the .NET library design. At runtime the Common Language Runtime knows the committed implementation that will be used throughout execution. However, compiled code does not itself pick up dependencies on the concrete representation.

☒ **Avoid** the use of implementation inheritance for extensibility.

In general, the .NET guidelines are quite agnostic with regard to the use of implementation inheritance. In F#, implementation inheritance is used more rarely than in other .NET languages. In F# there are many alternative techniques for designing and implementing object-oriented types using F#. Other object-oriented extensibility topics discussed in the .NET guidelines include events and callbacks, virtual members, abstract types and inheritance, and limiting extensibility by sealing classes.

3.3 Function and Member Signatures

☑ **Do** use tuples when appropriate for return values.

Here is a good example of using a tuple in a return type:

```
✓ val divrem : BigInteger -> BigInteger -> BigInteger * BigInteger
```

For return types containing many components, or where the components are related to a single identifiable entity, consider using a named type instead of a tuple.

☑ **Do** use `Async<T>` for async programming at F# API boundaries.
If there is a corresponding synchronous operation named `Foo` that returns a `T`, then the async operation should be named `AsyncFoo` and return `Async<T>`.
For commonly-used .NET types that expose Begin/End methods, consider using `Async.FromBeginEnd` to write extension methods as a façade to provide the F# async programming model to those .NET APIs.

```
✓ type SomeType =
    member this.Compute(x:int) : int = ...
    member this.AsyncCompute(x:int) : Async<int> = ...

✓ type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)
```

☑ **Consider** using option values for return types instead of raising exceptions (for F#-facing code).

The .NET approach to exceptions is that they should be “exceptional”; that is, they should occur relatively infrequently. However, some operations (for example, searching a table) may fail frequently. F# option values are an excellent way to represent the return types of these operations. These operations conventionally start with the name prefix “try”.

```
    // bad: throws exception if no element meets criteria
✗   member this.FindFirstIndex(pred : 'T -> bool) : int = ...

    // bad: returns -1 if no element meets criteria
✗   member this.FindFirstIndex(pred : 'T -> bool) : int = ...

    // good: returns None if no element meets criteria
✓   member this.TryFindFirstIndex(pred : 'T -> bool) : int option = ...
```

3.4 Exceptions

☑ **Do** follow the .NET guidelines for exceptions, including for F#-to-F# libraries.

The .NET Library Design Guidelines give excellent advice on the use of exceptions in the context of all .NET programming. Some of these guidelines are as follows:

- Do not return error codes. Exceptions are the main way of reporting errors in frameworks.
- Do not use exceptions for normal flow of control. Although this technique is often used in languages such as OCaml, it is bug-prone and can be inefficient on .NET. Instead consider returning a `None` option value to indicate failure.
- Do document all exceptions thrown by your components when a function is used incorrectly.
- Where possible throw existing exceptions from the `System` namespaces.
- Do not throw `System.Exception` when it will escape to user code. This includes avoiding the use of `failwith`, `failwithf`, which are handy functions for use in scripting and for code under development, but should be removed from F# library code in favor of throwing a more specific exception type.
- Do use `nullArg`, `invalidArg` and `invalidOp` as the mechanism to throw `ArgumentNullException`, `ArgumentException` and `InvalidOperationException` when appropriate.

3.5 Extension Members

☑ **Consider** using F# extension members in F#-to-F# code and libraries.

F# extension members should generally only be used for operations that are in the closure of intrinsic operations associated with a type in the majority of its modes of use. One common use is to provide APIs that are more idiomatic to F# for various .NET types:

```
✓ type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

✓ type System.Collections.Generic.IDictionary<'Key,'Value> with
      member this.TryGet key =
         let ok, v = this.TryGetValue key
         if ok then Some v else None
```

3.6 Union Types

☑ **Do** use discriminated unions as an alternative to class hierarchies for creating tree-structured data.

☑ **Do** use the `[<RequiredQualifiedAccess>]` attribute on union types whose case names are not sufficiently unique.

☑ **Consider** hiding the representations of discriminated unions for binary compatible APIs if the design of these types is likely to evolve.

Unions types rely on F# pattern-matching forms for a succinct programming model. As mentioned by previous guidelines you should avoid revealing concrete data representations such as records and unions in type design if the design of these types is likely to evolve.

For example, the representation of a discriminated union can be hidden using a `private` or `internal` declaration, or by using a signature file.

```
type Union =
   private
      | CaseA of int
      | CaseB of string
```

If you reveal discriminated unions indiscriminately, you may find it hard to version your library without breaking user code. You may consider revealing one or more active patterns to permit pattern matching over values of your type.

Active patterns provide an alternate way to provide F# consumers with pattern matching while avoiding exposing F# Union Types directly.

3.7 Inline Functions and Member Constraints

☑ **Consider** defining generic numeric algorithms using inline functions with implied member constraints and statically resolved generic types, for F#-facing code.

Arithmetic member constraints and F# comparison constraints are a highly regular standard for F# programming. For example, consider the following

```
let inline highestCommonFactor a b =
    let rec loop a b =
        if a = LanguagePrimitives.GenericZero<_> then b
        elif a < b then loop a (b - a)
        else loop (a - b) b
    loop a b
```

The type of this function is as follows:

```
val inline highestCommonFactor :
    ^T ->  ^T ->  ^T
    when ^T : (static member Zero : ^T)
    and  ^T : (static member ( - ) :  ^T *  ^T ->  ^T)
    and   ^T : equality
    and   ^T : comparison
```

This is a suitable function for a public API in a mathematical library.

☒ **Avoid** the over-using member constraints for other ad-hoc coding purposes in F# library designs.

It is possible to simulate “duck typing” using F# member constraints. However, members that make use of this should not in general be used in F#-to-F# library designs. This is because library designs based on unfamiliar or non-standard implicit constraints tend to cause user code to become inflexible and strongly tied to one particular framework pattern.

3.8 Operator Definitions

☒ **Avoid** defining custom symbolic operators in F#-facing library designs.

Custom operators are essential in some situations and are highly useful notational devices within a large body of implementation code. For new users of a library, named functions are often easier to use. In addition custom symbolic operators can be hard to document, and users find it more difficult to lookup help on operators, due to existing limitations in IDE and search engines.

As a result, it is generally best to publish your functionality as named functions and members.

3.9 Units of Measure

☑ **Do** use units of measure for added type safety in F# code, including in F#-facing libraries.

This type information is erased when viewed by other .Net languages, so be aware that .NET components, tools and reflection will just see types-sans-units (e.g. `float` rather than `float<kg>`) after this information has been erased.

3.10 Type Abbreviations

☑ **Consider** using type abbreviations to simplify F# code, including in F#-facing libraries.

Be aware that .NET components, tools and reflection will just see the types-being-abbreviated.

☒ **Avoid** using type abbreviations for public types whose members and properties should, logically speaking, be intrinsically different to those available on the type being abbreviated.

In this case, the type being abbreviated reveals too much about the representation of the actual type being defined. Instead, consider wrapping the abbreviation in a class type or a single-case discriminated union (or, when performance is absolutely essential, consider using a struct type to wrap the abbreviation).

For example, it is tempting to define a multi-map as a special case of an F# map, e.g.

```
type MultiMap<'Key,'Value> = Map<'Key,'Value list>
```

However, the logical dot-notation operations on this type are not the same as the operations on a Map – for example, it is reasonable that the lookup operator `map.[key]` return the empty list if the key is not in the dictionary, rather than raising an exception.

4 Guidelines for Libraries for Use from other .NET Languages

When designing libraries for use from other .NET languages, it is very important to adhere to the .NET Library Design Guidelines. In this document we label these libraries *vanilla .NET libraries*, as opposed to *F#-facing libraries* which use F# constructs without restriction and are mostly intended for use by F# applications. Designing vanilla .NET libraries means providing familiar and idiomatic APIs consistent with the rest of the .NET Framework by minimizing the use of F#-specific constructs in the public API. We propose the rules in the following sections.

4.1 Namespace and Type Design

☑ **Do** apply the .NET Library Design Guidelines to the public API of your components, for vanilla .NET APIs.

In particular, apply the naming guidelines, paying special attention to the use of abbreviated names and the .NET capitalization guidelines.

```
× type pCoord = ...
      member this.theta = ...

✓ type PolarCoordinate = ...
      member this.Theta = ...
```

☑ **Do** use namespaces, types and members as the primary organizational structure for your components (as opposed to modules), for vanilla .NET APIs.

This means all files containing public functionality should begin with a “namespace” declaration, and the only public-facing entities in namespaces should be types.

Use non-public modules to hold implementation code, utility types and utility functions.

Static types should be preferred over modules, as they allow for future evolution of the API to use overloading and other .NET API design concepts which may not be used within F# modules.

For example, in place of the following public API:

```
✗ module Fabrikom

  module Utilities =
     let Name = "Bob"
     let Add2 x y = x + y
     let Add3 x y z = x + y + z
```

Consider instead:

```
✓ namespace Fabrikom

   // A type in a component designed for use from other .NET languages
   [<AbstractClass; Sealed>]
    type Utilities =
     static member Name = "Bob"
     static member Add(x,y) = x + y
     static member Add(x,y,z) = x + y + z
```

☑ **Consider** using F# record types in vanilla .NET APIs if the design of the types will not evolve.

F# record types compile to a simple .NET class. These are suitable for some simple, stable types in APIs. You should consider using the `[<NoEquality>]` and `[<NoComparison>]` attributes to suppress the automatic generation of interfaces. Also avoid using mutable record fields in vanilla .NET APIs as these will expose a public field. Always consider whether a class would provide a more flexible option for future evolution of the API.

For example, this F# code will expose the public API below to a C# consumer.

F#:

```
[<NoEquality;NoComparison>]
type MyRecord =
    { FirstThing : int;
      SecondThing : string }
```

C#:

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

☑ **Do** hide the representation of F# union types in vanilla .NET APIs.

F# union types are not commonly used across component boundaries, even for F#-to-F# coding. They are an excellent implementation device when used internally within components and libraries.

When designing a vanilla .NET API, consider hiding the representation of a union type by using either a private declaration or a signature file.

```
✓ type PropLogic =
       private | And of PropLogic * PropLogic
               | Not of PropLogic
               | True
```

You may also augment types that use a union representation internally with members to provide a desired .NET-facing API. If the union representation is hidden, the corresponding methods and properties in the compiled form of the union type will also be hidden.

```
✓ type PropLogic =
     private | And of PropLogic * PropLogic
             | Not of PropLogic
             | True
     /// A public member for use from C#
     member x.Evaluate =
          match x with
          | And(a,b) -> a.Evaluate && b.Evaluate
          | Not a -> not a.Evaluate
          | True -> true

     /// A public member for use from C#
     static member MakeAnd(a,b) = And(a,b)
```

☑ **Consider** using the .NET code analysis tool FxCop with vanilla .NET APIs.

You can use this tool to check the public interface of your assembly for compliance with the .NET Library Design Guidelines. FxCop by default also checks some internal implementation properties which are not necessarily applicable to F# coding. As a result you may need to use FxCop exemptions where necessary.

☑ **Do** design GUI and other components using the design patterns of the particular .NET frameworks you are using. For example, for WPF programming, adopt WPF design patterns for the classes you are designing. For models in user interface programming, use design patterns such as events and notification-based collections such as those found in `System.Collections.ObjectModel`.

4.2 Object and Member Design

☑ **Do** use the CLIEvent attribute to expose .NET events, and construct a `DelegateEvent` with a specific .NET delegate type that takes an object and `EventArgs` (rather than an `Event`, which just uses the `FSharpHandler` type by default) so that the events are published in the familiar way to other .NET languages.

```
× type MyBadType() =
    let myEv = new Event<int>()
    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

✓ /// A type in a component designed for use from other .NET languages
  type MyEventArgs(x:int) =
    inherit System.EventArgs()
    member this.X = x

  /// A type in a component designed for use from other .NET languages
  type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()
    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

☑ **Do** expose asynchronous operations using either the .NET asynchronous programming model (BeginFoo, EndFoo), or as methods returning .NET tasks (Task<T>), rather than as F# Async<T> objects.

The Asynchronous Programming Model (APM) is a standard pattern for asynchronous APIs on .NET. An F# `Async<T>` computation can be exposed as an APM pattern using the F# `Async.AsBeginEnd` method. This method makes it easy to convert an F# async object into Begin/End/Cancel methods. Consider also exposing the Cancel method, in addition to the Begin/End methods that are the necessary portion of the pattern.

```
✓ // A type in a component designed for use from other .NET languages
  type MyType() =
    let compute(x:int) : Async<int> = async { ... }
    let beginAction, endAction, cancelAction =
        Async.AsBeginEnd (fun x -> compute x)
    member this.BeginCompute(x, callback, state:obj) =
        beginAction(x, callback, state)
    member this.EndCompute(result) = endAction result
    member this.CancelCompute(result) = cancelAction result
```

In .NET 4.0, the Task model was introduced, and tasks can represent active asynchronous computations. Tasks are in general less compositional than F# Async<T> objects, since they represent "already executing" tasks and can't be composed together in ways that perform parallel composition, or which hide the propagation of cancellation signals and other contextual parameters.

However, despite this, methods-returning-Tasks are emerging as a standard representation of asynchronous programming on .NET.

```
✓ // A type in a component designed for use from other .NET languages
  type MyType() =
    let compute(x:int) : Async<int> = async { ... }
    member this.ComputeAsTask(x) = compute x |> Async.StartAsTask
```

You will frequently also want to accept an explicit cancellation token:

```
✓ /// A type in a component designed for use from other .NET languages
  type MyType() =
    let compute(x:int) : Async<int> = async { ... }
    member this.ComputeAsTask(x,token) = Async.StartAsTask(compute x, token)
```

☑ **Do** use .NET delegate types in preference to F# function types in vanilla .NET APIs.

Here "F# function types" mean "arrow" types like "int -> int".

```
× member this.Transform(f:int->int) = ...

✓ member this.Transform(f:Func<int,int>) = ...
```

The F# function type appears as "`class FSharpFunc<T,U>`" to other .NET languages, and is less suitable for language features and tooling that understand delegate types. When authoring a higher-order method targeting .NET 3.5 or higher, the `System.Func` and `System.Action` delegates are the right APIs to publish to enable .NET developers to consume these APIs in a low-friction manner. (When targeting .NET 2.0, the system-defined delegate types are more limited; consider using `System.Converter<T,U>` or defining a specific delegate type.)

On the flip side, .NET delegates are *not* natural for F#-facing libraries (see the previous section on F#-facing libraries). As a result, a common implementation strategy when developing higher-order methods for vanilla .NET libraries is to author all the implementation using F# function types, and then create the public API using delegates as a thin façade atop the actual F# implementation.

☑ **Consider** use the TryGetValue pattern instead of returning F# option values (`option<T>`) in vanilla .NET APIs, and prefer method overloading to taking F# option values as arguments.

Common patterns of use for the F# option type in APIs are better implemented in vanilla .NET APIs using standard .NET design techniques. Instead of returning an F# option value, consider using the bool return type plus an `out` parameter as in the TryGetValue pattern. And instead of taking F# option values as parameters, consider using method overloading.

```
× member this.ReturnOption() = Some 3
✓ member this.ReturnBoolAndOut(outVal : byref<int>) =
      outVal <- 3
      true

× member this.ParamOption(x : int, y : int option) =
      match y with | Some y' -> x + y' | None -> x
✓ member this.ParamOverload(x : int) = x
✓ member this.ParamOverload(x : int, y : int) = x + y
```

☑ **Consider** using the .NET collection interface types `IEnumerable<T>` and `IDictionary<Key,Value>` for parameters and return values in vanilla .NET APIs.

This means avoid the use of concrete collection types such as .NET arrays `T[]`, F# types `list<T>`, `Map<Key,Value>` and `Set<T>` , and.NET concrete collection types such as `Dictionary<Key,Value>`. The .NET Library Design Guidelines have good advice regarding when to use various collection types like `IEnumerable<T>`. Some use of arrays (`T[]`) is acceptable in some circumstances, on performance grounds. Note especially that `seq<T>` is just the F# alias for `IEnumerable<T>`, and thus `seq` is often an appropriate type for a vanilla .NET API.

```
× member this.PrintNames(names : list<string>) = ...

✓ member this.PrintNames(names : seq<string>) = ...
```

☑ **<u>Do</u>** use the unit type as the only input type of a method to define a zero-argument method, or as the only return type to define a void-returning method.

Avoid other uses of the unit type.

```
✓ member this.NoArguments() = 3
✓ member this.ReturnVoid(x : int) = ()
× member this.WrongUnit( x:unit, z:int) = ((), ())
```

☑ **<u>Consider</u>** checking for null values on vanilla .NET API boundaries.

F# implementation code tends to have fewer null values, due to immutable design patterns and restrictions on use of null literals for F# types. Other .NET languages often use null as a value much more frequently. Because of this, F# code that is exposing a vanilla .NET API should check parameters for null at the API boundary, and prevent these values from flowing deeper into the F# implementation code.

You can check for null using:

```
let checkNonNull argName (arg: obj) =
    match arg with
    | null -> nullArg argName
    | _ -> ()

static member ExampleMethod(record:obj,info:string) =
    checkNonNull "info" info
    checkNonNull "record" record
    ...
```

☒ **<u>Avoid</u>** using tuples as return values in vanilla .NET APIs.
Instead, prefer returning a named type holding the aggregate data, or using `out` parameters to return multiple values. Although Tuples are now part of .NET4.0, they will most often not provide the ideal and expected API for .NET developers. C# and VB programmers who get a Tuple return value will need to use properties like .Item1 and .Item2 to access the elements of the Tuple.

☒ **<u>Do not</u>** use currying of parameters in vanilla .NET APIs.

Instead, use .NET calling conventions Method(arg1,arg2,…,argN). Curried methods will appear in .NET as methods which return F# function values. As earlier guidance indicates, these types should not be exposed on vanilla .NET APIs, so curried members should be avoided in these APIs.

✓ member this.TupledArguments(str, num) = String.replicate num str

× member this.CurriedArguments str num = String.replicate num str

---

Tip If you're designing libraries for use from any .NET language, then there's no substitute for actually doing some experimental C# and Visual Basic programming to ensure that your libraries look good from these languages. You can also use tools such as .NET Reflector and the Visual Studio Object Browser to ensure that libraries and their documentation appear as expected to developers.

---

5 Recommendations for Implementation Code

In this section, we look at a small number of recommendations when writing implementation code (as opposed to library designs). These apply to all `internal` and `private` aspects of F# coding.

5.1 Suggested Naming Conventions in F# Implementation Code

☑ **Suggest** using camelCase for class-bound, expression-bound and pattern-bound values and functions

It is common and accepted F# style to use camelCase for all names bound as local variables or in pattern matches and function definitions.

```
✗ let add I J = I+J
✓ let add i j = i + j
```

It is also common and accepted F# style to use camelCase for locally bound functions, e.g.

```
✓ type MyClass() =
       let doSomething () =
           let firstResult = ...
           let secondResult = ...

       member x.Result = doSomething()
```

☑ **Suggest** using camelCase for internal and private module-bound values and functions

It is common and accepted F# style to use camelCase for private module-bound values, including the following:

- Ad hoc functions in scripts
- Values making up the internal implementation of a module or type

```
✓ let emailMyBossTheLatestResults = ...
```

In large assemblies and implementation files, PascalCase is sometimes used to indicate major internal routines.

☒ **Avoid** using underscores in names.

Historically, some F# libraries have used underscores in names. However, this is no longer widely accepted, partly because it clashes with .NET naming conventions. That said, some F# programmers use underscores heavily, partly for historical reasons, and tolerance and respect is important. However, be aware that the style is often disliked by others who have a choice about whether to use it.

Note Some F# programmers choose to use naming conventions much more closely associated with OCaml, Python, or with a particular application domain such as hardware verification.

5.2 Suggested Coding Conventions in F# Implementation Code

☑ **Do** use standard F# operators

The following operators are defined in the F# standard library and should be used instead of defining equivalents. Using these operators is recommended as it tends to make code more readable and idiomatic. Developers with a background in OCaml or other functional programming language may be accustomed to different idioms, and this list summarizes the recommended F# operators.

```
x |> f    -- forward pipeline
f <| x    -- reverse pipeline
f >> g    -- forward composition
g << f    -- reverse composition

x |> ignore   -- throwing away a value

x + y     -- overloaded addition (including string concatenation)
x - y     -- overloaded subtraction
x * y     -- overloaded multiplication
x / y     -- overloaded division
x % y     -- overloaded modulus

x && y    -- lazy/short-cut "and"
x || y    -- lazy/short-cut "or"

x <<< y  -- bitwise left shift
x >>> y  -- bitwise right shift
x ||| y  -- bitwise or, also for working with “flags” enumeration
x &&& y  -- bitwise and, also for working with “flags” enumeration
x ^^^ y  -- bitwise xor, also for working with “flags” enumeration
```

☑ **Do** place pipeline operator `|>` at the start of a line when writing multi-line pipeline series.

People often ask how to format pipelines. We recommend this style:

```
let allTypes =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> Array.map (fun assem -> assem.GetTypes())
    |> Array.concat
```

Note that this does not apply when piping on a single line (e.g. "`expr |> ignore`", "`expr |> box`")

☑ **Consider** using the prefix syntax for generics (`Foo<T>`) in preference to postfix syntax (`T Foo`), with four notable exceptions (`list`, `option`, `array`, `ref`).

F# inherits both the postfix ML style of naming generic types, e.g. "`int list`" as well as the prefix .NET style, e.g. "`list<int>`". You should prefer the .NET style, except for four specific types. For F# lists, use the postfix form: "`int list`" rather than "`list<int>`". For options, use the postfix form: "`int option`" rather than "`option<int>`". For arrays, use the syntactic name "`int[]`" rather than either "`int array`" or "`array<int>`". For refs, use "`int ref`" rather than "`ref<int>`" or "`Ref<int>`". For all other types, use the prefix form: "`HashSet<int>`", "`Dictionary<string,int>`", since this conforms to .NET standards.

☑ **Consider** using `//` comments in preference to `(*…*)` comments.

Line comments `//` are easier to see in code because they appear consistently at the beginning of lines. They are also typically more predictable when reading code line by line. Tools like Visual Studio also make it particularly easy to comment/uncomment whole blocks with `//`.

6 Appendix

6.1 End-to-end example of designing F# code for use by other .NET languages

For example, consider the code in Listing 1, which shows some F# code that we intend to adjust to be suitable for use as part of a .NET API.

Listing 1. An F# Type Prior to Adjustment for Use as Part of a Vanilla .NET API

```
open System
type Point1(angle,radius) =
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
    new() = Point1(angle=0.0, radius=0.0)
```

The inferred F# type of this class is as follows:

```
type Point1 =
     new : unit -> Point1
     new : angle:double * radius:double -> Point1
     static member Circle : n:int -> Point1 list
     member Stretch : l:double -> Point1
     member Warp : f:(double -> double) -> Point1
     member Angle : double
     member Radius : double
```

Let's take a look at how this F# type will appear to a programmer using another .NET language. For example, the approximate C# "signature" is as follows:

```
// C# signature for the unadjusted Point1 class

public class Point1 {
     public Point1();
     public Point1(double angle, double radius);
     public static Microsoft.FSharp.Collections.List<Point1> Circle(int
count);
     public Point1 Stretch(double factor);
     public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double>
transform);
     public double Angle { get; }
     public double Radius  { get; }
}
```

There are some important points to notice about how F# represents constructs here. For example:

- Metadata such as argument names has been preserved.
- F# methods that take two arguments become C# methods that take two arguments.
- Functions and lists become references to corresponding types in the F# library.

The full rules for how F# types, modules, and members are represented in the .NET Common Intermediary Language are explained in the F# language reference on the F# website.

The code below shows how to adjust this code to take these things into account.

```
namespace ExpertFSharp.Types

type RadialPoint(angle:double, radius:double) =

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,_>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
                  RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)
```

The inferred F# type of the code is as follows:

```
type RadialPoint =
     new : unit -> RadialPoint
     new : angle:double * radius:double -> RadialPoint
     static member Circle : count:int -> seq<RadialPoint>
     member Stretch : factor:double -> RadialPoint
     member Warp : transform:System.Func<double,double> -> RadialPoint
     member Angle : double
     member Radius : double
```

The C# signature is now as follows:

```
public class RadialPoint {
   public RadialPoint();
   public RadialPoint(double angle, double radius);
   public static System.Collections.Generic.IEnumerable<RadialPoint>
                Circle(int count);
   public RadialPoint Stretch(double factor);
   public RadialPoint Warp(System.Func<double,double> transform);
   public double Angle { get; }
   public double Radius  { get; }
}
```

The fixes we have made to prepare this type for use as part of a vanilla .NET library are as follows:

- We adjusted several names; `Point1` , `n`, `l`, and `f` became `RadialPoint`, `count`, `factor`, and `transform`, respectively.
- We used a return type of `seq<RadialPoint>` instead of `RadialPoint list` by changing a list construction using `[ ... ]` to a sequence construction using `seq { ... }`.
- We used the .NET delegate type `System.Func` instead of an F# function type.