

Circularities and Modularity in the Wild

Some F# Perspectives on Software Engineering

Don Syme

(Microsoft Research)

Scott Wlaschin

(fpbridge.co.uk, fsharpforfunandprofit.com)

Agenda

- A bit about F#
- Circularity and Modularity In The Wild
- Tooling, Data and Type Providers

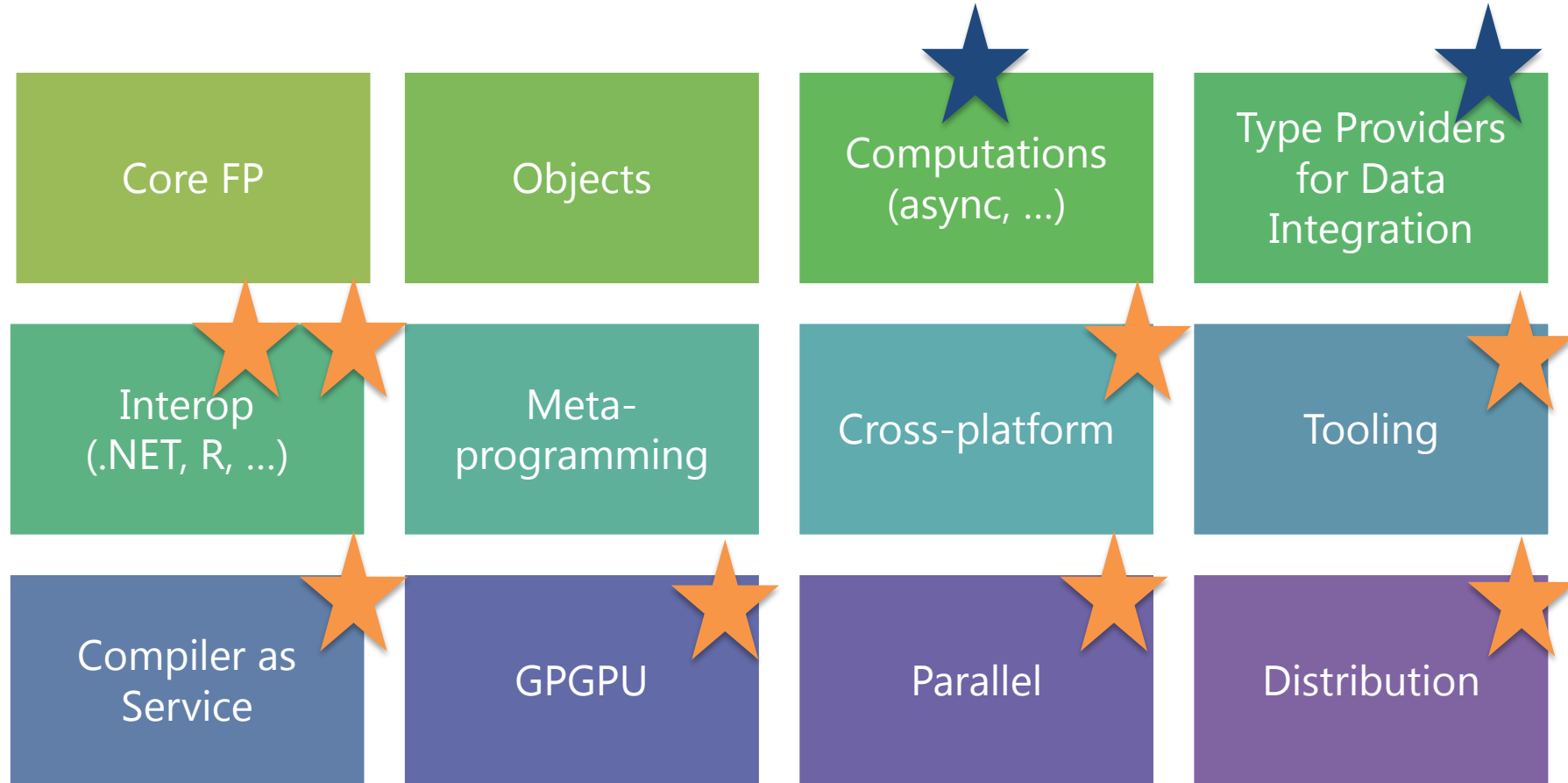
F# is open source, cross-platform,
community-oriented

fsharp.org

#fsharp on Twitter

meetup.com/FSharpLondon
(on tonight!)

For this audience, F# is....



github.com/fsharp/fsharp
github.com/fsprojects/VisualFSharpPowerTools

github.com/fsharp/FSharp.Compiler.Service
github.com/fsharp/FSharp.Data

F# helps address real business problems for real businesses

Time to Market, Efficiency, Correctness and Complexity

See: "Succeeding with Functional-first Programming in Industry"

See fsharp.org/testimonials

F# has changed...

"F# is for Windows"



F# runs on
many
platforms

F# has changed...

"Microsoft
makes F#"




F# has many
contributors

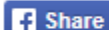
F# has changed...

One perspective
(Microsoft's)
<http://msdn.microsoft.com>




Many
perspectives
<http://fsharp.org>



 Share

103

 Tweet

693

About

- [About F#](#)
- [Learning F#](#)
- [Development guide](#)
- [Documentation](#)
- [Language spec](#)
- [Testimonials](#)

Getting F#

- [F# on Mac](#)
- [F# on Linux](#)
- [F# on Windows](#)
- [F# on Android](#)
- [F# on iOS \(iPhone/iPad\)](#)

The F# Software Foundation

F# is a mature, open source, cross-platform, functional-first programming language which empowers users and organizations to tackle complex computing problems with simple, maintainable and robust code.

F# runs on Linux, Mac OS X, Android, iOS, Windows as well as HTML5 and GPUs. F# is free to use and has an OSI-approved open source license.

F# is used in a wide range of application areas and is supported by both industry-leading companies providing professional tools, and by an active open community.


The F# Software Foundation exists to promote, protect, and advance F#, and to support and foster the growth of a diverse international community of F# users.

First Steps with F#

- View [F# testimonials](#)
- Learn on [Try F#](#)
- Explore [F# books and](#)

Going Further with F#

- Ask F# questions on [StackOverflow](#)
- Find [F# meetups](#)

 Follow @fsharporg

Application Areas

- [Data Science](#)
- [Web Programming](#)
- [Apps and Games](#)
- [Machine Learning](#)
- [Cloud Programming](#)
- [Financial Computing](#)
- [Math and Statistics](#)
- [Data Access](#)

Contributors to F#

- [The F# Software Foundation](#)
- [Xamarin](#)
- [Microsoft](#)
- [Training Companies](#)
- [more](#)

Circularities and Modularity in the Wild

Analysis by Scott Wlaschin and F# Community

Mixed OO/Functional Programming Has Won

Lambdas in C#, Java, C++, ...

Async monadic modality in C#, Javascript, PHP (Hack), ...

Function types in C#, TypeScript, ...

Generics in C#, Java, Visual Basic, ...

...

... except where it hasn't

Inheritance everywhere!!!

Nulls everywhere!!!

Side effects everywhere!!!!

Circularities everywhere!!!!

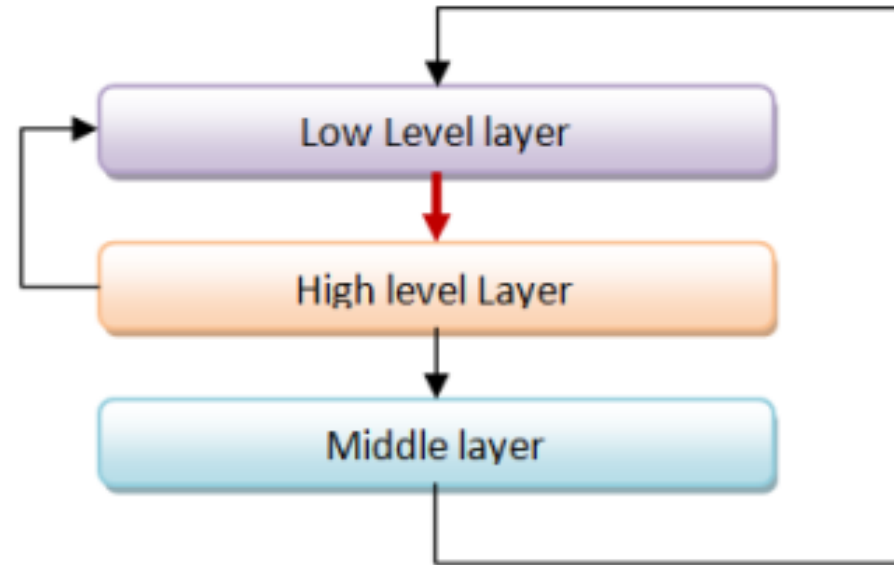
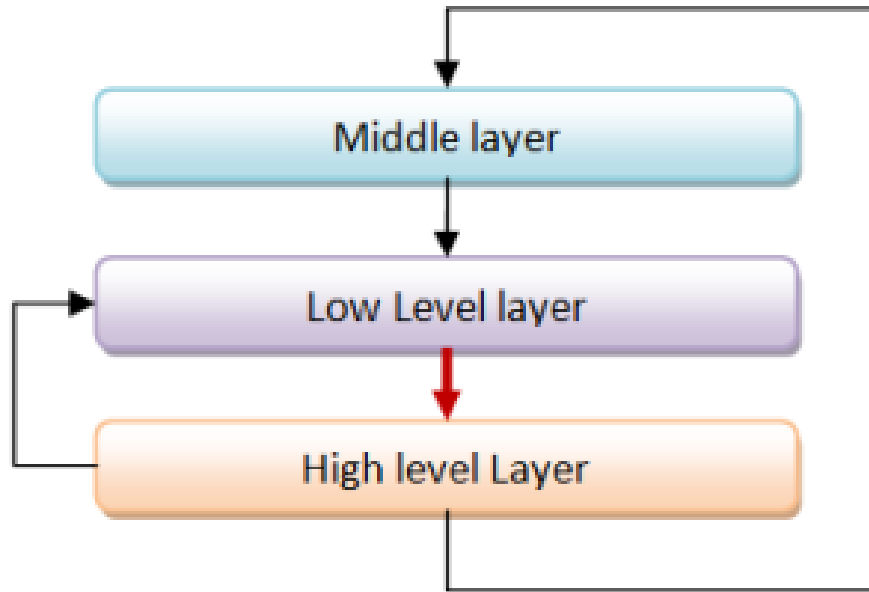
F# and Software Engineering

- By accident, F# makes for an interesting, large-scale experiment in SE
- Same runtime, JIT, GC, standard libraries, IDE, machines as C#
- "FP v. OO Cage Match of Death" (though not really....)

Q: Does the structure of F# code differ from C# in practice?

Q: Does the language you use make a difference?

Unnecessary Circularities are Evil



- To the functional programmer, it is “obvious” that our software methodology should help minimize and reduce cyclic dependencies in program structure.

The C# approach to circularity

- All files in an “assembly” are mutually referential
- Arbitrary circularity and dependency between “internal” items in an assembly
- [InternalsVisibleTo] can reveal internals of one assembly to another
- Mutually recursive “assemblies” are possible if you try hard

The F# approach to circularity

- Like Haskell and all Hindley-Milner languages, F# has a file ordering
- F# prohibits direct circularities across files
- F# encourages minimizing dependencies within a file
- Parameterization the preferred technique
- F# objects support recursion and limited forms of circularity when needed

Let's analyse some C# and F# projects

C# projects

- [Mono.Cecil](#), which inspects programs and libraries in the ECMA CIL format.
- [NUnit](#)
- [SignalR](#) for real-time web functionality.
- [NancyFx](#), a web framework
- [YamlDotNet](#), for parsing and emitting YAML.
- [SpecFlow](#), a BDD tool.
- [Json.NET](#).
- [Entity Framework](#).
- [ELMAH](#), a logging framework for ASP.NET.
- [NuGet](#) itself.
- [Moq](#), a mocking framework.
- [NDepend](#), a code analysis tool.
- And, to show I'm being fair, a business application that I wrote in C#.

Let's analyse some C# and F# projects

F# projects

- [FSharp.Core](#), the core F# library.
- [FSPowerPack](#).
- [FsUnit](#), extensions for NUnit.
- [Canopy](#), a wrapper around the Selenium test automation tool.
- [FsSql](#), a nice little ADO.NET wrapper.
- [WebSharper](#), the web framework.
- [TickSpec](#), a BDD tool.
- [FSharpX](#), an F# library.
- [FParsec](#), a parser library.
- [FsYaml](#), a YAML library built on FParsec.
- [Storm](#), a tool for testing web services.
- [Foq](#), a mocking framework.
- Another business application that I wrote, this time in F#.

Sizes of projects

Project	Code size	Top-level types	Authored types	All types
ef	269521	514	565	876
jsonDotNet	148829	215	232	283
nancy	143445	339	366	560
cecil	101121	240	245	247
nuget	114856	216	237	381
signalR	65513	192	229	311
nunit	45023	173	195	197
specFlow	46065	242	287	331
elmah	43855	116	140	141
yamlDotNet	23499	70	73	73
fparsecCS	57474	41	92	93
moq	133189	397	420	533
ndepend	478508	734	828	843
ndependPlat	151625	185	205	205
personalCS	422147	195	278	346
TOTAL	2244670	3869	4392	5420

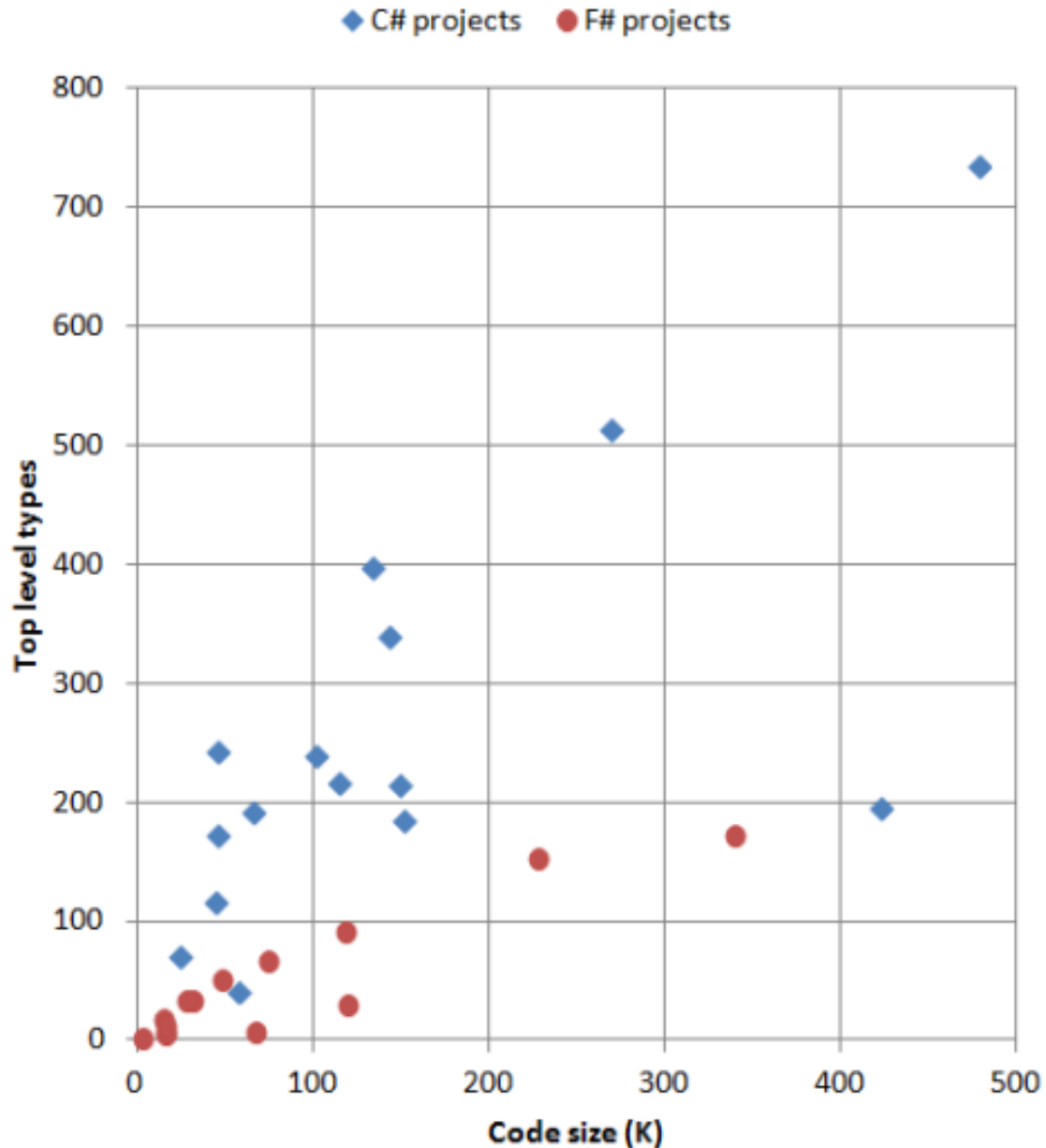
Project	Code size	Top-level types	Authored types	All types
fsxCore	339596	173	328	2024
fsCore	226830	154	313	1186
fsPowerPack	117581	93	150	410
storm	73595	67	70	405
fParsec	67252	8	24	245
websharper	47391	52	128	285
tickSpec	30797	34	49	170
websharperHtml	14787	18	28	72
canopy	15105	6	16	103
fsYaml	15191	7	11	160
fsSql	15434	13	18	162
fsUnit	1848	2	3	7
foq	26957	35	48	103
personalFS	118893	30	146	655
TOTAL	1111257	692	1332	5987

Metrics Used

- A *top-level type* is: a type that is not nested and which is not compiler generated.
- Roughly speaking, a count of user-defined types and modules
- Code size is measured by IL instructions (not LOC)

How many top-level type definitions or modules?

Code size vs. # of top level types



F# tends to have fewer type definitions or modules.

C# projects have ~1-2 TTT/1K instructions.

F# projects have ~0.6 TTT/1K instructions.

Reasons:

(a) functional abstraction is used more often

(b) functions, discriminated unions etc. mean fewer type definitions and modules

What about dependencies?

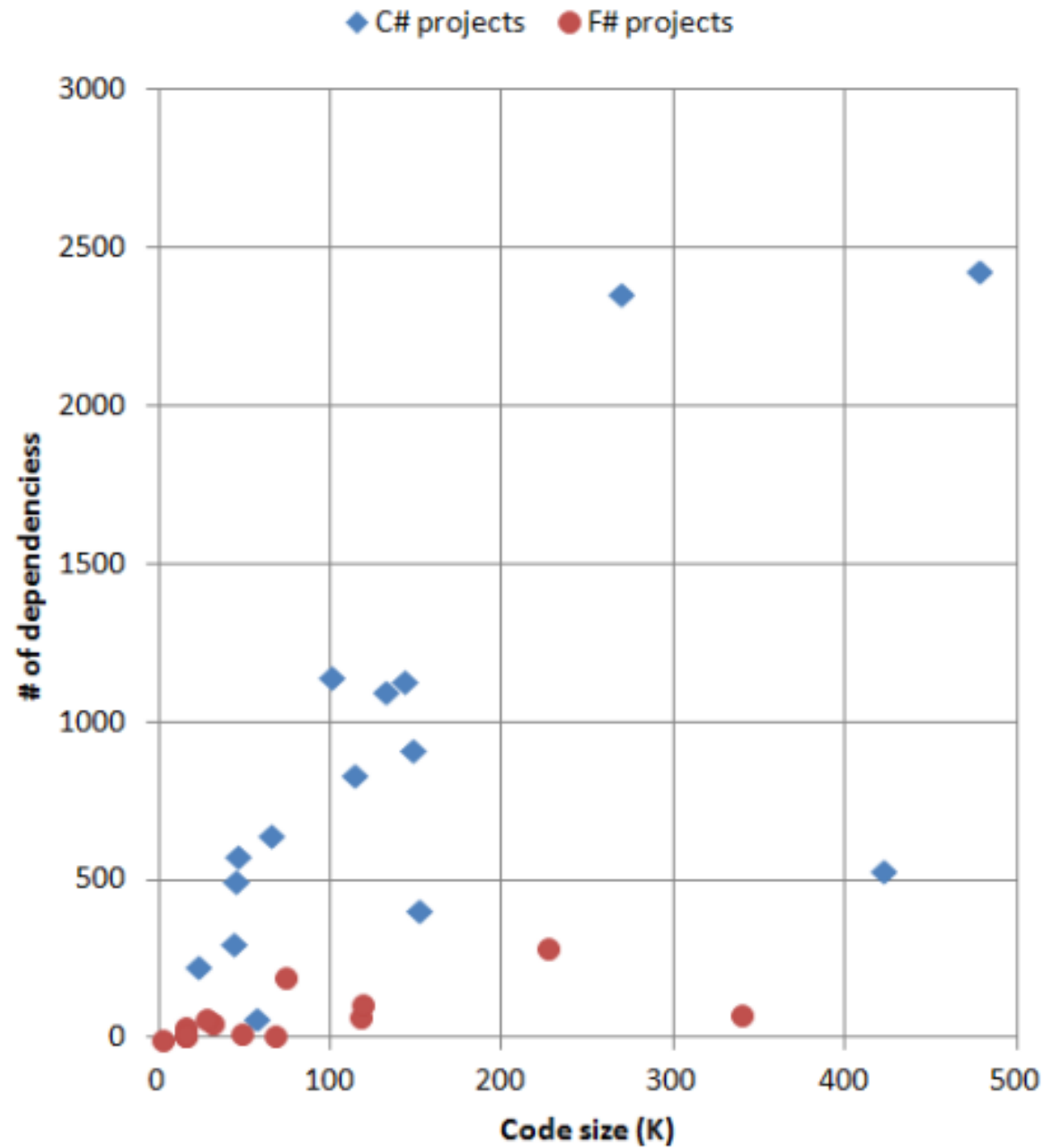
Definition of dependency

Let's say we have a top-level type `A` and another top-level type `B`. Then I say that a *dependency* exists from `A` to `B` if:

- Type `A` or any of its nested types inherits from (or implements) type `B` or any of its nested types.
- Type `A` or any of its nested types has a field, property or method that references type `B` or any of its nested types as a parameter or return value. This includes private members as well -- after all, it is still a dependency.
- Type `A` or any of its nested types has a method implementation that references type `B` or any of its nested types.

This might not be a perfect definition. But it is good enough for my purposes.

Code size vs. # of dependencies



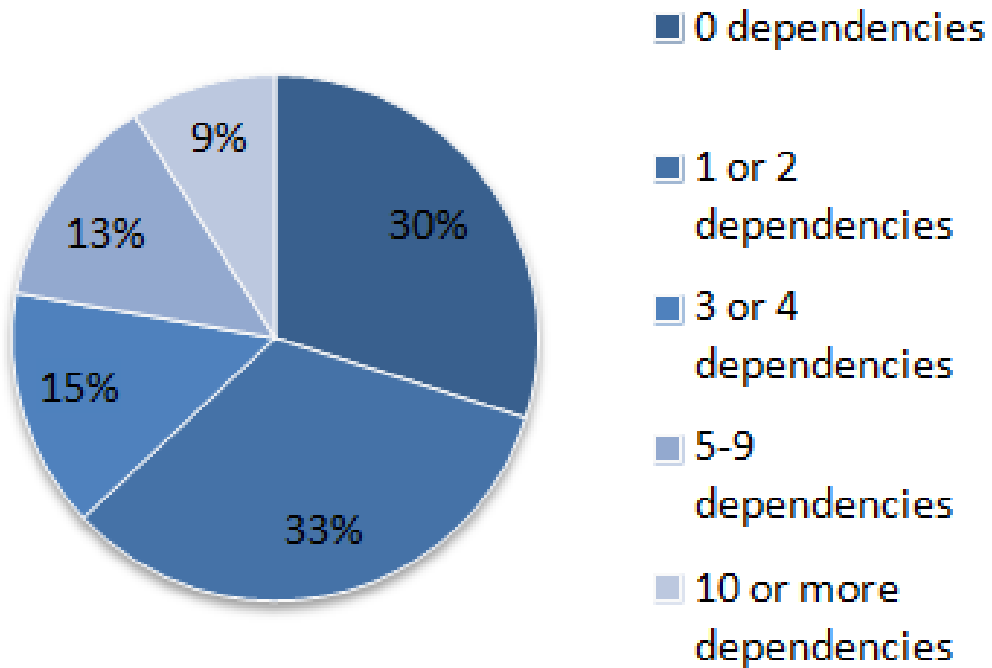
F# code has fewer top-level-type/module dependencies

For C#, the number of total dependencies increases with project size. Each top-level type depends on 3-4 others, on average

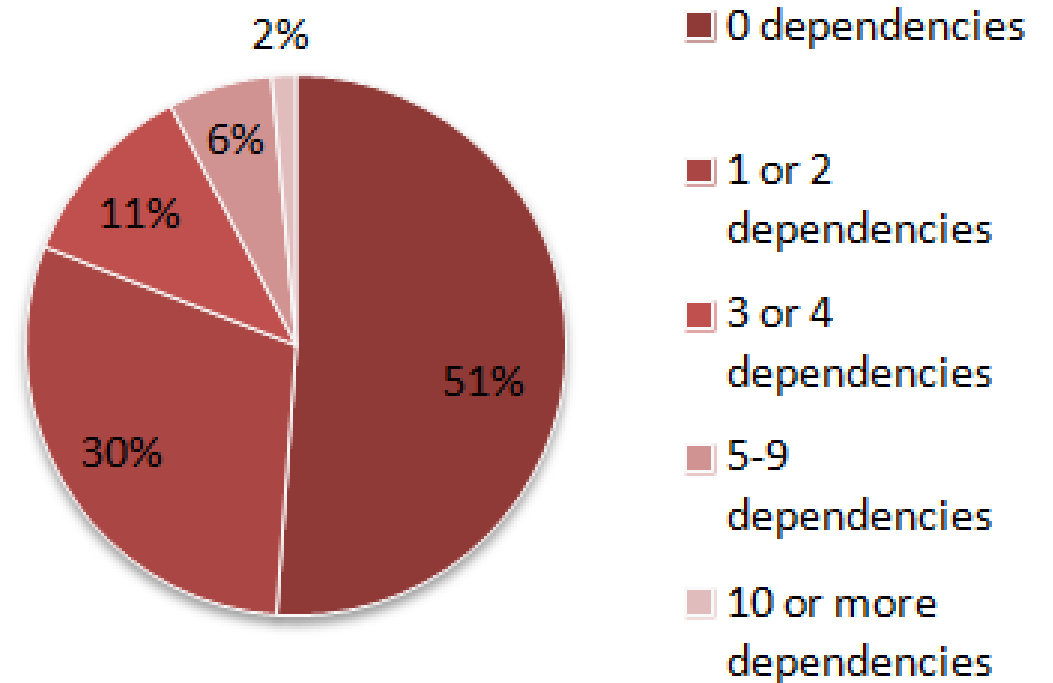
For F# each F# type/module depends on no more than 1-2 others, on average

Let's take a look at the distribution of dependencies...

C# projects
Top level types grouped
by number of dependencies



F# projects:
Top level types grouped
by number of dependencies



Here are the results for the C# projects:

Project	Top Level Types	Total Dep. Count	Dep/Top	One or more dep.	Three or more dep.	Five or more dep.	Ten or more dep.
ef	514	2354	4.6	76%	51%	32%	13%
jsonDotNet	215	913	4.2	69%	42%	30%	14%
nancy	339	1132	3.3	78%	41%	22%	6%
cecil	240	1145	4.8	73%	43%	23%	13%
nuget	216	833	3.9	71%	43%	26%	12%
signalR	192	641	3.3	66%	34%	19%	10%
nunit	173	499	2.9	75%	39%	13%	4%
specFlow	242	578	2.4	64%	25%	17%	5%
elmah	116	300	2.6	72%	28%	22%	6%
yamlDotNet	70	228	3.3	83%	30%	11%	4%
fparsecCS	41	64	1.6	59%	29%	5%	0%
moq	397	1100	2.8	63%	29%	17%	7%
ndepend	734	2426	3.3	67%	37%	25%	10%
ndependPlat	185	404	2.2	67%	24%	11%	4%
personalCS	195	532	2.7	69%	29%	19%	7%
TOTAL	3869	13149	3.4	70%	37%	22%	9%

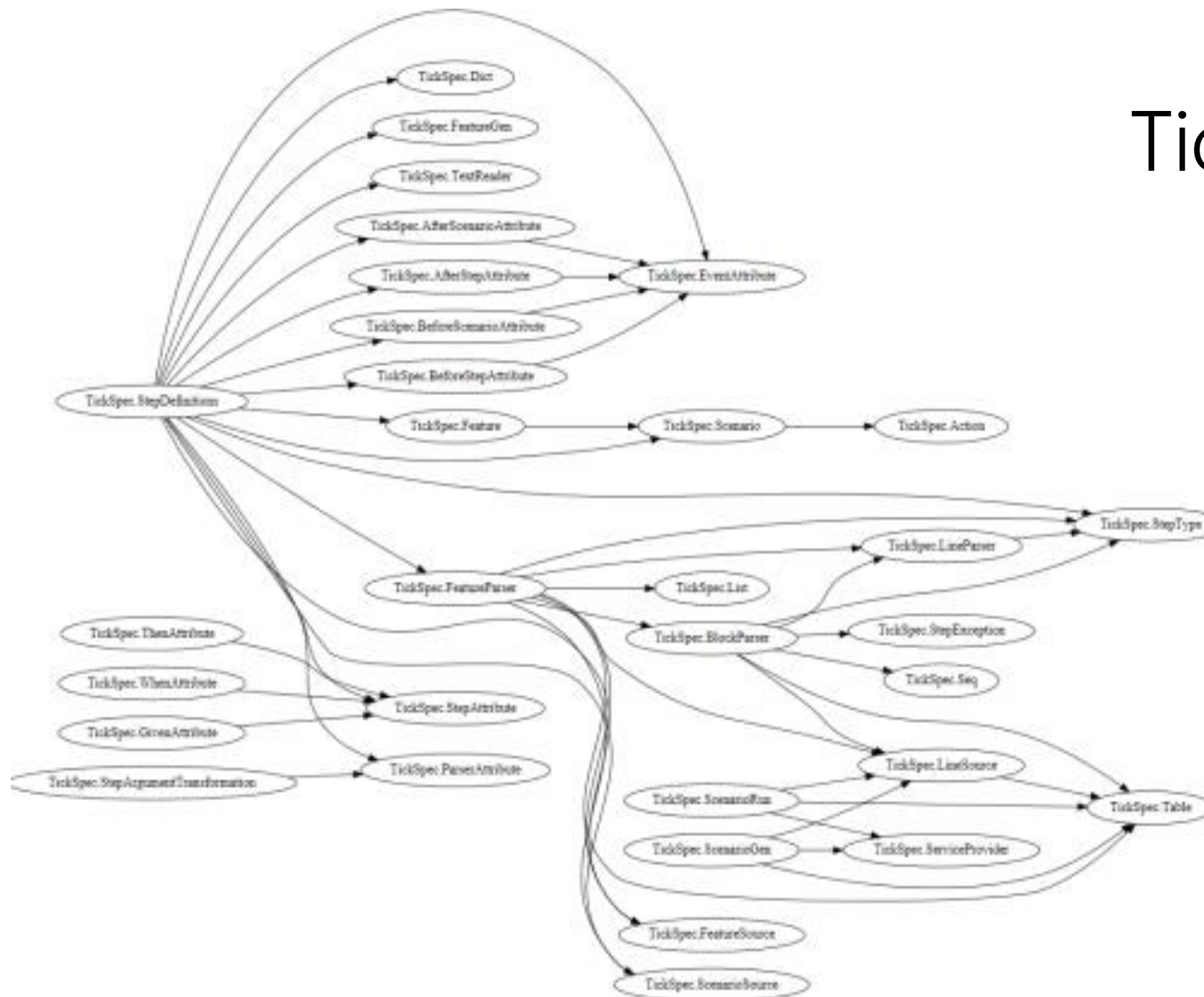
Project	Top Level Types	Total Dep. Count	Dep/Top	One or more dep.	Three or more dep.	Five or more dep.	Ten or more dep.
fsxCore	173	76	0.4	30%	4%	1%	0%
fsCore	154	287	1.9	55%	26%	14%	3%
fsPowerPack	93	68	0.7	38%	13%	2%	0%
storm	67	195	2.9	72%	40%	18%	4%
fParsec	8	9	1.1	63%	25%	0%	0%
websharper	52	18	0.3	31%	0%	0%	0%
tickSpec	34	48	1.4	50%	15%	9%	3%
websharperHtml	18	37	2.1	78%	39%	6%	0%
canopy	6	8	1.3	50%	33%	0%	0%
fsYaml	7	10	1.4	71%	14%	0%	0%
fsSql	13	14	1.1	54%	8%	8%	0%
fsUnit	2	0	0.0	0%	0%	0%	0%
foq	35	66	1.9	66%	29%	11%	0%
personalFS	30	111	3.7	93%	60%	27%	7%
TOTAL	692	947	1.4	49%	19%	8%	1%

Let's compare two similar projects...



SpecFlow (F#)

TickFlow (F#)



Similar feature set – the code is organized differently. It highlights some of the differences between OO design and functional design.

SpecFlow is well designed, and a useful library. It uses good OOD and TDD practices:

- TestRunnerManager, ITestRunnerManager
- "listener" classes and interfaces
- "provider" classes and interfaces
- "comparer" classes and interfaces.....

TickSpec uses no interfaces at all:

- no "listeners", "providers" or "comparers"
- where needed the role they play is fulfilled by functions

What about circularity?

Methodology

Doing the experiment

First, I downloaded each of the project binaries using NuGet. Then I wrote a little F# script that did the following steps for each assembly:

1. Analyzed the assembly using [Mono.Cecil](#) and extracted all the types, including the nested types
2. For each type, extracted the public and implementation references to other types, divided into internal (same assembly) and external (different assembly).
3. Created a list of the "top level" types.
4. Created a dependency list from each top level type to other top level types, based on the lower level dependencies.

This dependency list was then used to extract various statistics, shown below. I also rendered the dependency graphs to SVG format (using [graphViz](#)).

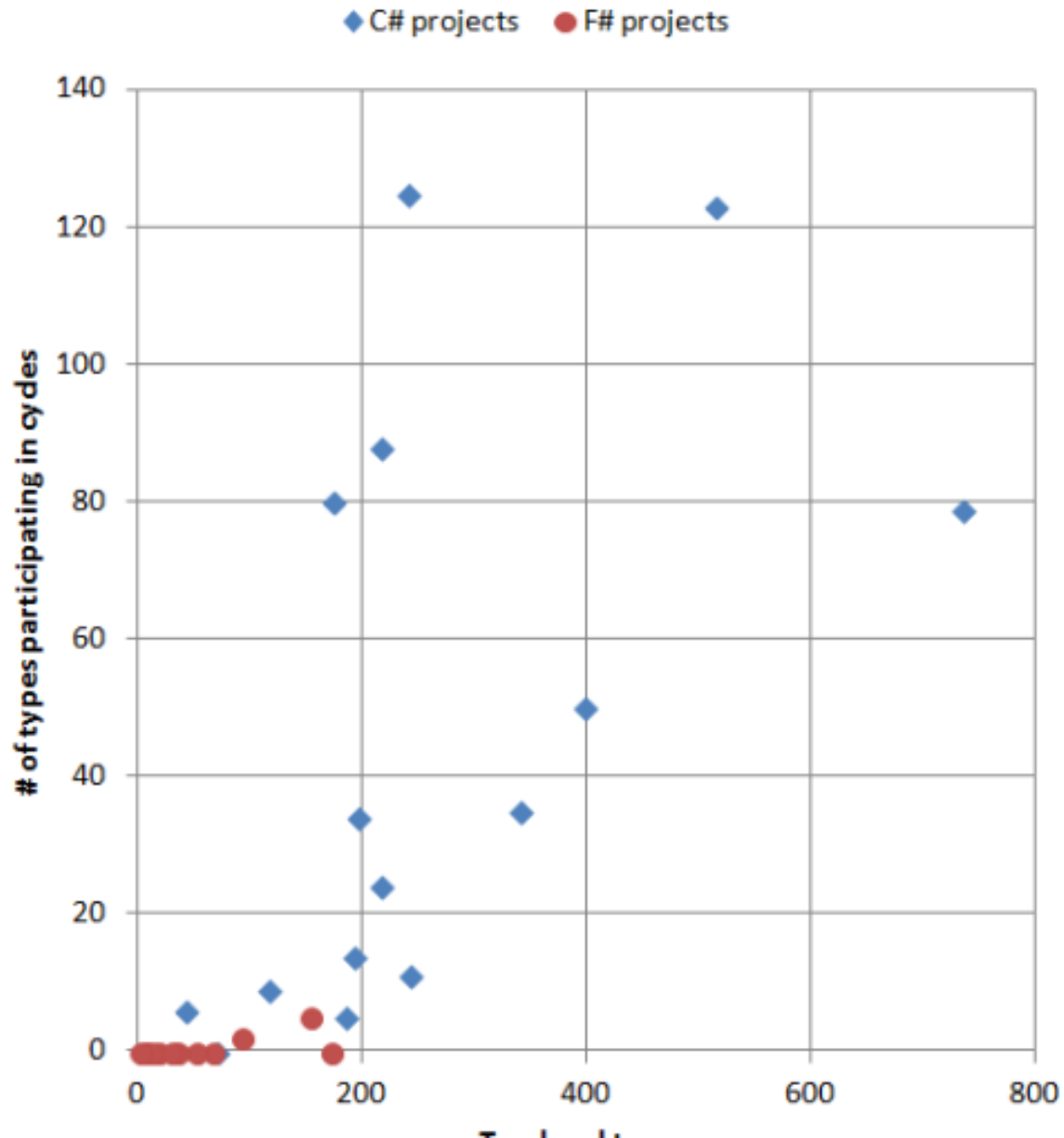
For cycle detection, I used the [QuickGraph library](#) to extract the strongly connected components, and then did some more processing and rendering.

If you want the gory details, here is [a link to the script](#) that I used, and [here is the raw data](#).

Project	Top-level types	Cycle count	Partic.	Partic.%	Max comp. size	Cycle count (public)	Partic. (public)	Partic.% (public)	Max comp. size (public)
ef	514	14	123	24%	79	1	7	1%	7
jsonDotNet	215	3	88	41%	83	1	11	5%	11
nancy	339	6	35	10%	21	2	4	1%	2
cecil	240	2	125	52%	123	1	50	21%	50
nuget	216	4	24	11%	10	0	0	0%	1
signalR	192	3	14	7%	7	1	5	3%	5
nunit	173	2	80	46%	78	1	48	28%	48
specFlow	242	5	11	5%	3	1	2	1%	2
elmah	116	2	9	8%	5	1	2	2%	2
yamlDotNet	70	0	0	0%	1	0	0	0%	1
fparsecCS	41	3	6	15%	2	1	2	5%	2
moq	397	9	50	13%	15	0	0	0%	1
ndepend	734	12	79	11%	22	8	36	5%	7
ndependPlat	185	2	5	3%	3	0	0	0%	1
personalCS	195	11	34	17%	8	5	19	10%	7
TOTAL	3869		683	18%			186	5%	

Project	Top-level types	Cycle count	Partic.	Partic.%	Max comp. size	Cycle count (public)	Partic. (public)	Partic.% (public)	Max comp. size (public)
fsxCore	173	0	0	0%	1	0	0	0%	1
fsCore	154	2	5	3%	3	0	0	0%	1
fsPowerPack	93	1	2	2%	2	0	0	0%	1
storm	67	0	0	0%	1	0	0	0%	1
fParsec	8	0	0	0%	1	0	0	0%	1
websharper	52	0	0	0%	1	0	0	0%	0
tickSpec	34	0	0	0%	1	0	0	0%	1
websharperHtml	18	0	0	0%	1	0	0	0%	1
canopy	6	0	0	0%	1	0	0	0%	1
fsYaml	7	0	0	0%	1	0	0	0%	1
fsSql	13	0	0	0%	1	0	0	0%	1
fsUnit	2	0	0	0%	0	0	0	0%	0
foq	35	0	0	0%	1	0	0	0%	1
personalFS	30	0	0	0%	1	0	0	0%	1
TOTAL	692		7	1%			0	0%	

Top level types vs. participation in cycles



Why the difference between C# and F#?

In C#, there is nothing stopping you from creating cycles. In fact, you have to make a special effort to avoid them.

In F#, you can't easily create cycles at all.

Circularity as it really is...



(and that's just $1/4$ of the graph...)

Conclusion....

In theory and in practice, unmoderated intra-assembly cycles are a disaster

Language mechanisms that enforce layering are necessary and good

The really hard question:

Unmoderated cycles are like crack

How do we help an utterly addicted industry?

(Whether via OO, FP or other languages?)

A related analysis (Simon Cousins, Energy Sector)

350,000

lines of C# OO
by offshore team

The C# project took five years and peaked at ~8 devs. It never fully implemented all of the contracts.

The F# project took less than a year and peaked at three devs (only one had prior experience with F#). All of the contracts were fully implemented.

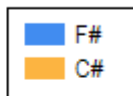
30,000

lines of robust F#, with
parallel + more features

An application to evaluate the revenue due from [Balancing Services](#) contracts in the UK energy industry

<http://simontcousins.azurewebsites.net/does-the-language-you-use-make-a-difference-revisited/>

2000

**Implementation****C#****F#****Braces**

56,929

643

Blanks

29,080

3,630

Null Checks

3,011

15

Comments

53,270

487

Useful Code

163,276

16,667

App Code

305,566

21,442

Test Code

42,864

9,359

Total Code

G

348,430

30,801

0

try

Other F# Topics

F# Basics

F# for Data
Science

F# for GPUs

F# for Cloud
Data

F# for Testing

F# for DSLs

F# + R

F# Data Integration through Type providers

Questions?

F# is open source, cross-platform,
community-oriented

fsharp.org

meetup.com/FSharpLondon

#fsharp on Twitter

In Summary

Open, cross-platform,
strongly typed, efficient,
rock-solid stable

The safe choice for
functional-first

F#

Unbeatable data integration

Visual F# - tooling you can
trust from Microsoft