ROADMAP

- **What is F#**
- A quick introduction
- Does the language make a difference?
- What is doing F# differently?
- Adoption?
- Challenges?

WHAT IS F#

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

fsharp.org

WHAT IS F#

"F# was so easy to pick up we went from complete novices to having our code in production in less than a week."

Jack Mott from O'Connor's Online

---

fsharp.org/testimonials

WHAT IS F#

"Most successful projects I have written have all been in F#."

Colin Bull, talking about enterprise software

WHAT IS F#

- Open Source
- Functional-First -> multi-paradigm
- Less error prone
  - No Nulls by default
  - Stongly typed
  - Units of measure
- Expressive
  - Pattern matching: Records, Tuples, Discriminated Unions
  - Scripting -> Automation (even one-liners)
  - Active Patterns
  - Triple-quoted strings
  - Object expressions

WHAT IS F#

- Less Work
  - Let the compiler do the work
  - Code reusability
  - More declarative
  - Quickly try several solutions
- Meta programming
  - Transpiler to Javascript (FunScript, Fable), and GPU code
  - Quotations
  - Type providers -> JSON (+ REST Apis, for example WorldBank), XML, PowerShell, Python, "R", SQL, Registry, WMI, FileSystem, HTML, Excel, CSV
  - Computation Expressions -> async, sequence, cloud, asyncSeq or your own

WHAT IS F#

Just another .NET language

ROADMAP

- What is F#
- **A quick introduction**
- Does the language make a difference?
- What is doing F# differently?
- Adoption?
- Challenges?

F# IS REALLY SIMPLE

```
var t = 5;
```

```
let u = 5
```

- No semi colons required
- u is immutable by default (a symbol no variable)

F# IS REALLY SIMPLE

```
public int Add(int a, int b) {
   return a + b;
}
```

```
let add a b =
   a + b
```

- No return keyword, last value is returned.
- No types needed, compiler will figure it out.
- No start- and end-brace is needed, whitespace counts.

F# IS REALLY SIMPLE

```csharp
public class Person {
  public string Name { get; } // C#7
  public Person(string name) { Name = name; }
}
```

```fsharp
type Person (name:string) =
  member x.Name = name
```

- No start- and end-brace is needed, whitespace counts.
- Single constructor by default (helps you design better classes)
- Constructor parameters are private fields out of the box

F# IS REALLY SIMPLE

```
public interface IPerson {
  public string Name { get; }
}
```

```
type IPerson =
  abstract Name : string
```

- Interface = Type without implementations and without constructor
- You can do everything you would expect: abstract classes, namespaces, public, private, internal, ...

DISCRIMINATED UNIONS AND PATTERN MATCHING

```fsharp
type Fruit =
  | Apple of radius:int
  | Banana of length:int

let printFruit fruit =
  match fruit with
  | Apple radius -> sprintf "a tasty apple with radius %d" radius
  | Banana length -> sprintf "banana with size %d" length

```

- C#: 3 classes are required (abstract base class Fruit, Apple, Banana)
- Visitor pattern
- Equality members
- Documentation
- printFruit is another class (the visitor)

F# IS SIMPLE: MATCH = SWITCH ON STEROIDS

```
var a = o as A;
if (a != null)
{
    //...
}
var b = o as B;
if (b != null && someCondition)
{
    // ...
}

```

```
match o with
| :? A as a -> //...
| :? B as b when someCondition -> //...
```

F# IS SIMPLE: CONSISTENT SYNTAX

```
try {
  // ...
}
catch (AException a) {
  // ...
}
catch (BException b) {
  if (!someCondition) throw;
  // ...
}
```

```
try
  // ...
with
| :? AException as a -> //...
| :? BException when someCondition -> //...
```

F# IS SIMPLE: RECORDS

```
public class Person {
  public string Name { get; }
  public string Address { get; }
  public Person(string name, string address) {
    Name = name; Address = address } }
```

```
type Person =
  { Name : string; Address : string option }

let createPerson name = { Name = name; Address = None }
let printPerson p =
    match p with
    | { Name = "Lars" } -> "the boss"
    | _ -> p.Name
```

- Equality members.
- Pattern matching.
- Immutable by default.

ROADMAP

- What is F#
- A quick introduction
- **Does the language make a difference?**
- What is doing F# differently?
- Adoption?
- Challenges?

DOES THE LANGUAGE MAKE A DIFFERENCE?

- The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.
- The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

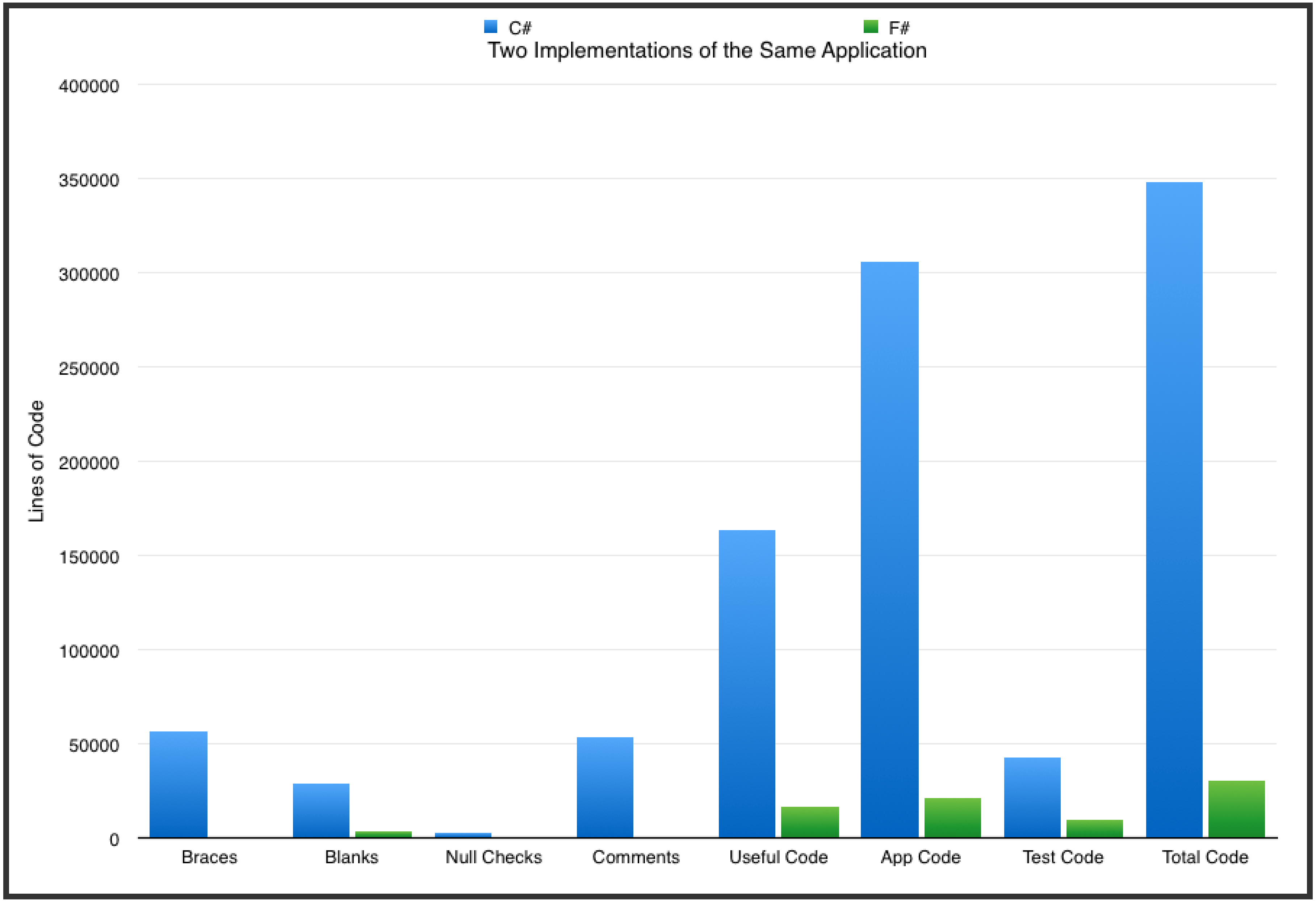
Edsger W. Dijkstra

DOES F# MAKE A DIFFERENCE?

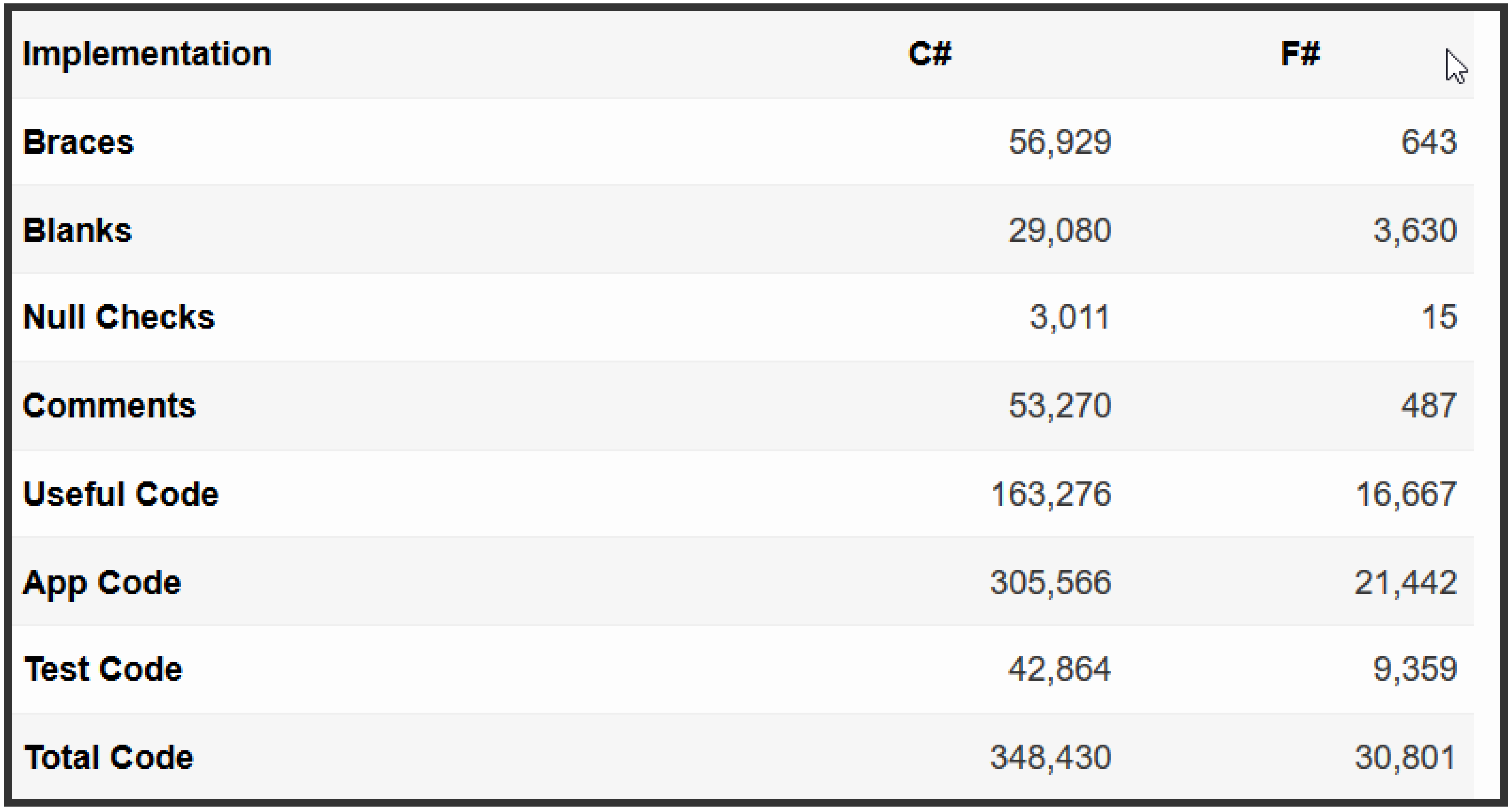
What do people say about C# and F#?

- Real enterprise system
- Two different teams
- Same set of contracts, complex contracts
- Analysis afterwards

DOES F# MAKE A DIFFERENCE?



DOES F# MAKE A DIFFERENCE?



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	348,430	30,801

DOES F# MAKE A DIFFERENCE?

- 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.

F# makes a difference

ROADMAP

- What is F#
- A quick introduction
- Does the language make a difference?
- **What is doing F# differently?**
- Adoption?
- Challenges?

WHAT IS DOING F# DIFFERENTLY?

- Layout, Readability
- Naming
- Understandable and Expressive
- Abstractions

LAYOUT: WHITESPACE

```
public int Method() {
    return 3;
}
```

Or

```
public int Method ()
{
    return 3;
}
```

Two competing rules in C-like languages

LAYOUT: WHITESPACE

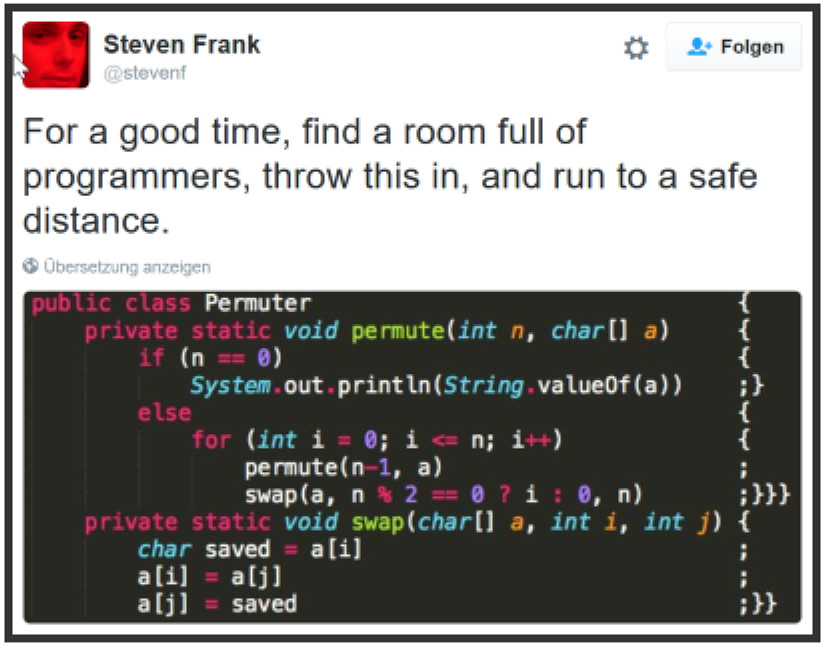
There should be one - and preferable only one - obvious way to do it.

the Zen of Python

LAYOUT: WHITESPACE

Language	**Compiler**	**Human**
C#	{}	{} and whitespace
F#	whitespace	whitespace

LAYOUT: WHITESPACE



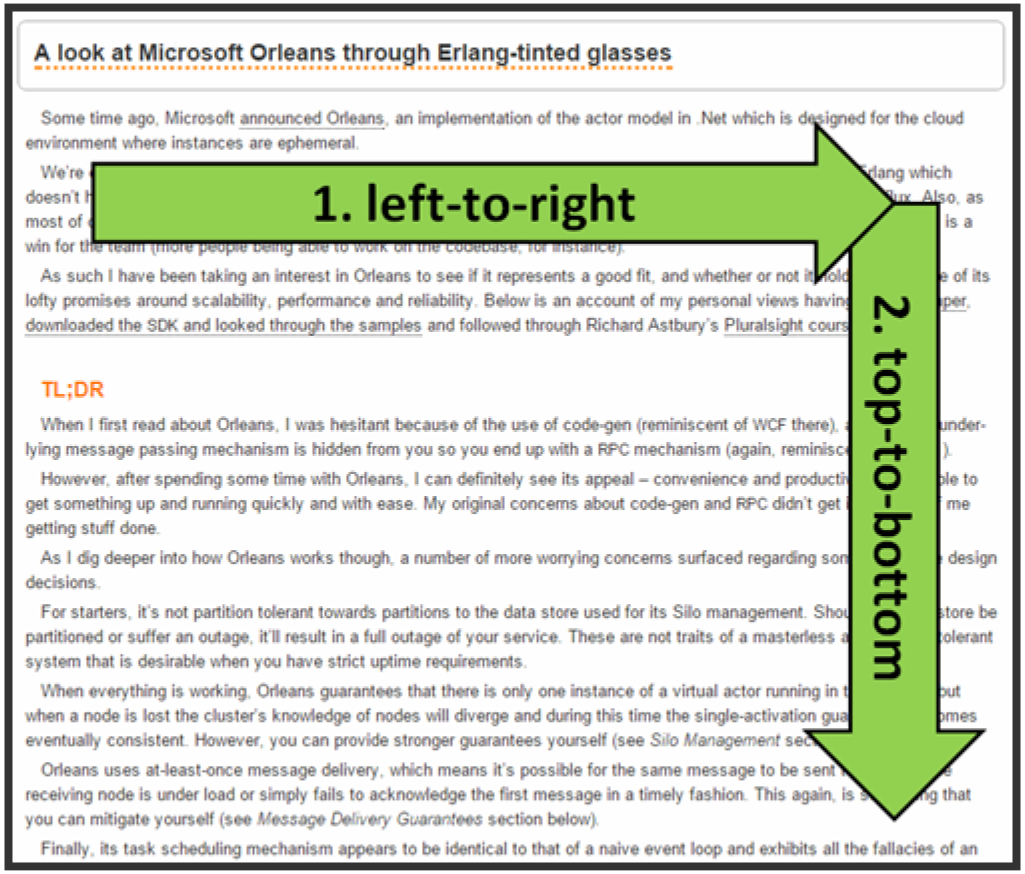
Steven Frank
@stevenf

For a good time, find a room full of programmers, throw this in, and run to a safe distance.

Übersetzung anzeigen

```
public class Permuter                                              {
    private static void permute(int n, char[] a)                   {
        if (n == 0)                                                {
            System.out.println(String.valueOf(a))                  ;}
        else                                                       {
            for (int i = 0; i <= n; i++)                           {
                permute(n-1, a)                                    ;
                swap(a, n % 2 == 0 ? i : 0, n)                     ;}}}
    private static void swap(char[] a, int i, int j)               {
        char saved = a[i]                                          ;
        a[i] = a[j]                                                ;
        a[j] = saved                                               ;}}
```

LAYOUT: STRUCTURE



A look at Microsoft Orleans through Erlang-tinted glasses

Some time ago, Microsoft announced Orleans, an implementation of the actor model in .Net which is designed for the cloud environment where instances are ephemeral.

We're lang which doesn't h lux. Also, as most of is a win for the team (more people being able to work off the codebase, for instance).

As such I have been taking an interest in Orleans to see if it represents a good fit, and whether or not it hold e of its lofty promises around scalability, performance and reliability. Below is an account of my personal views having per, downloaded the SDK and looked through the samples and followed through Richard Astbury's Pluralsight cours

TL;DR

When I first read about Orleans, I was hesitant because of the use of code-gen (reminiscent of WCF there), under-lying message passing mechanism is hidden from you so you end up with a RPC mechanism (again, reminisc).

However, after spending some time with Orleans, I can definitely see its appeal – convenience and producti ble to get something up and running quickly and with ease. My original concerns about code-gen and RPC didn't get me getting stuff done.

As I dig deeper into how Orleans works though, a number of more worrying concerns surfaced regarding son design decisions.

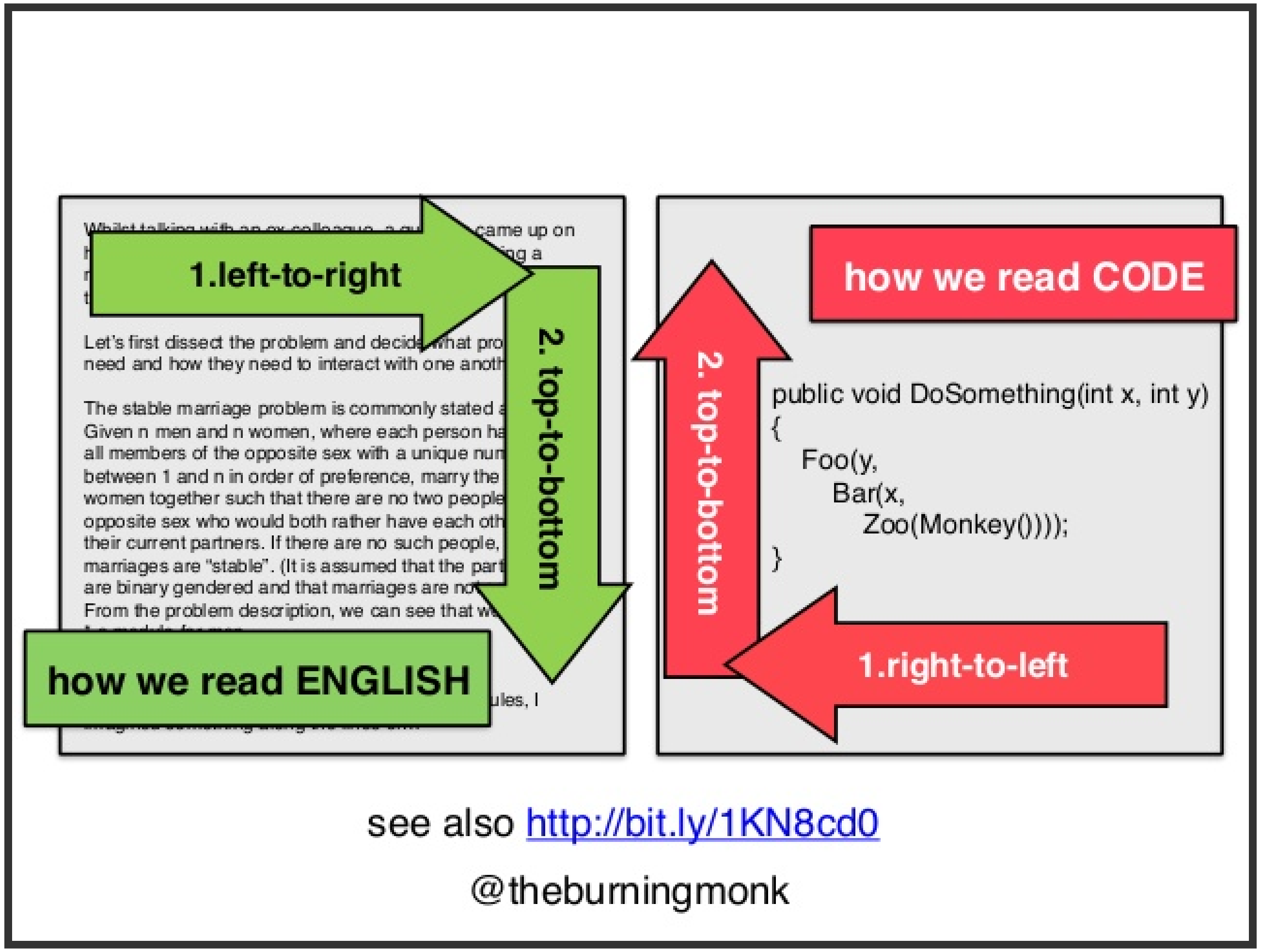
For starters, it's not partition tolerant towards partitions to the data store used for its Silo management. Shou store be partitioned or suffer an outage, it'll result in a full outage of your service. These are not traits of a masterless a tolerant system that is desirable when you have strict uptime requirements.

When everything is working, Orleans guarantees that there is only one instance of a virtual actor running in t but when a node is lost the cluster's knowledge of nodes will diverge and during this time the single-activation gua omes eventually consistent. However, you can provide stronger guarantees yourself (see *Silo Management* sec

Orleans uses at-least-once message delivery, which means it's possible for the same message to be sent e receiving node is under load or simply fails to acknowledge the first message in a timely fashion. This again, is ng that you can mitigate yourself (see *Message Delivery Guarantees* section below).

Finally, its task scheduling mechanism appears to be identical to that of a naive event loop and exhibits all the fallacies of an

LAYOUT: STRUCTURE



Let's first dissect the problem and decide what pro need and how they need to interact with one anoth

The stable marriage problem is commonly stated a Given n men and n women, where each person ha all members of the opposite sex with a unique num between 1 and n in order of preference, marry the women together such that there are no two people opposite sex who would both rather have each oth their current partners. If there are no such people, marriages are "stable". (It is assumed that the part are binary gendered and that marriages are no From the problem description, we can see that w

```
public void DoSomething(int x, int y)
{
    Foo(y,
        Bar(x,
            Zoo(Monkey())));
}
```

see also http://bit.ly/1KN8cd0

@theburningmonk

LAYOUT: STRUCTURE

```
let doSomething x y =
    Monkey()
     |> zoo
     |> bar x
     |> foo y

```

NAMING

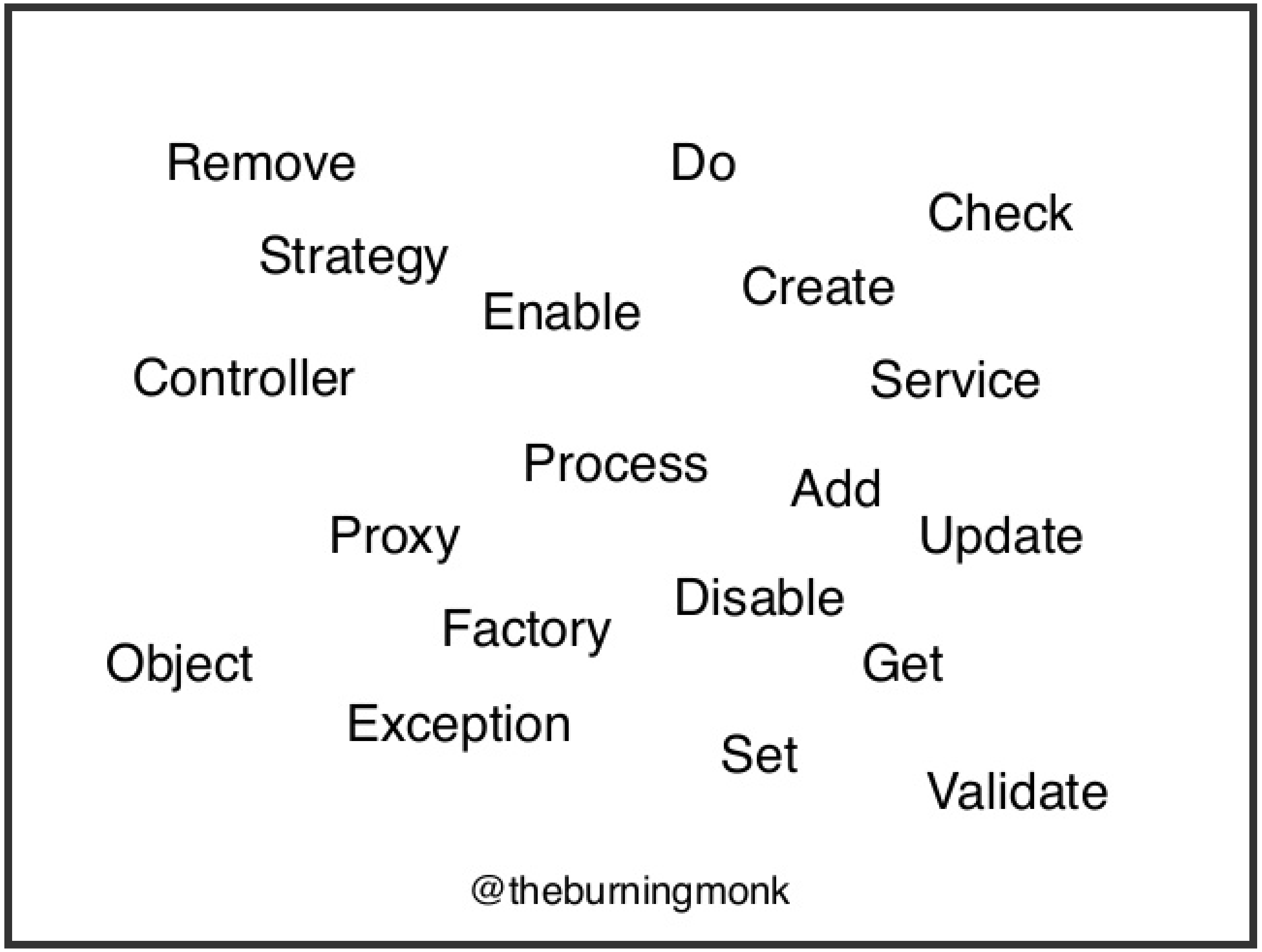
"There are only two hard things in Computer Science: cache invalidation and naming things."

Phil Karlton

"Names are the one and only tool you have to explain what a variable does in every place it appears, without having to scatter comments everywhere."

Mike Mahemoff

LEGO NAMING



methodnamer.com

NAMING: HIGHER ORDER FUNCTIONS

```
words
|> Array.map (fun x -> x.Count)
|> Array.reduce (+)
```

- Smaller scopes
- Shorter names (When x, y, z are great variable names)
- Fewer things to name

NAMING: SHORT NAMES

"The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names."

***Clean Code**: Robert C. Martin*

NAMING: SMALLER THAN OBJECT?

```
public interface IConditionChecker
{
	bool CheckCondition();
}
```

```
public interface ICondition
{
	bool IsTrue();
}
```

NAMING: SMALLER THAN OBJECT?

```
type Condition = unit -> bool
```

```
using Condition = System.Func<System.Boolean>;
```

No abstraction is too small.

NAMING: OBJECT EXPRESSIONS

```
enterpriseCrew.OrderBy(
  (fun c -> c.Current),
  { new IComparer<Occupation> with
      member __.Compare(x, y) =
        x.Position.CompareTo(y.Position) })
```

- No need to define a class, no need to name it.
- `_` and `__` used to explicitly give something no name.

EXPRESSIVENESS: "EVERYTHING" IS AN EXPRESSION

```
var variable = null;
try {
  variable = TrySomethingRisky()
} catch (AException) {
  variable = fallback1;
} catch (BException) {
  variable = fallback2;
}
return DoSomething(variable);

```

```
let symbol =
  try
    TrySomethingRisky()
  with
    | :? AException -> fallback1
    | :? BException -> fallback1
DoSomething(symbol)

```

EXPRESSIVENESS: "EVERYTHING" IS AN EXPRESSION

```
let symbol =
  try
    TrySomethingRisky()
  with
    | :? AException -> fallback1
    | :? BException -> fallback1
DoSomething(symbol)

```

```
try
  TrySomethingRisky()
with
  | :? AException -> fallback1
  | :? BException -> fallback1
|> DoSomething

```

- No need to name the thing.

EXPRESSIVENESS: "EVERYTHING" IS AN EXPRESSION

```
var variable = condition ? Value1 : fallback;
```

```
let variable = if condition then Value1 else fallback
```

SINGLE PASS COMPILER

DEMO

EXPRESSIVENESS: TYPE PROVIDER

DEMO

EXPRESSIVENESS: COMPUTATION EXPRESSION

```
async {
    let! results =
        [ "http://www.mbrace.io/"
          "http://www.nessos.gr/" ]
        |> List.map downloadAsync
        |> Async.Parallel

    return results |> Array.sumBy(fun r -> r.Length)
}
```

Like async/await in C# but the concept behind is more powerful.

EXPRESSIVENESS: COMPUTATION EXPRESSION

```
let downloadCloud url = downloadAsync url |> Cloud.OfAsync

cloud {
    let! results =
        [ "http://www.mbrace.io/"
          "http://www.nessos.gr/" ]
        |> List.map downloadCloud
        |> Cloud.Parallel

    return results |> Array.sumBy(fun r -> r.Length)
}
```

mbrace

ABSTRACTIONS

"Your abstractions should afford right behaviour, whilst make it impossible to do the wrong thing."

"Make illegal states unrepresentable."

ABSTRACTIONS: MAKE ILLEGAL STATES UNREPRESENTABLE

Business rules:

- A contact has a name.
- A contact has an address.
    - an email address
    - a postal address
    - both

ABSTRACTIONS: MAKE ILLEGAL STATES UNREPRESENTABLE

```
public class Contact {
  public string Name { get; }
  public EmailAddress EMail { get; }
  public PostalAddress Address { get; }
  public Contact(string name, EmailAddress email,
      PostalAddress address) {
    if (email == null && address == null)
      throw new ArgumentException("Invalid!");
    Name = name;
    EMail = email;
    Address = address;
}
```

ABSTRACTIONS: MAKE ILLEGAL STATES UNREPRESENTABLE

```fsharp
type ContactInfo =
    | EmailOnly of EmailAddress
    | PostOnly of PostalAddress
    | EmailAndPost of EmailAddress * PostalAddress

type Contact = { Name : string; ContactInfo : ContactInfo }
```

ROADMAP

- What is F#
- A quick introduction
- Does the language make a difference?
- What is doing F# differently?
- **Adoption?**
- Challenges?

ADOPTION?

Low risk

- Runs on CLR, mono and the new "dotnetcore"
- Open Source
- Good Interop
- Back-out to C#

@simontcousins

ADOPTION?

- Self taught
- Hire .NET developers, not language X
- Production code in a week
- Functional programmer in a month

@simontcousins

ADOPTION?

- Baby steps: Don't try to introduce a new language and a new paradigm
- Language first: Explicit interfaces, Syntax, Records vs Classes vs Modules
- Paradigm second: Let the language guide you

Colin Bull

ADOPTION?

- Start with Build
- msbuild is not fun
- FAKE

Colin Bull

ROADMAP

- What is F#
- A quick introduction
- Does the language make a difference?
- What is doing F# differently?
- Adoption?
- **Challenges?**

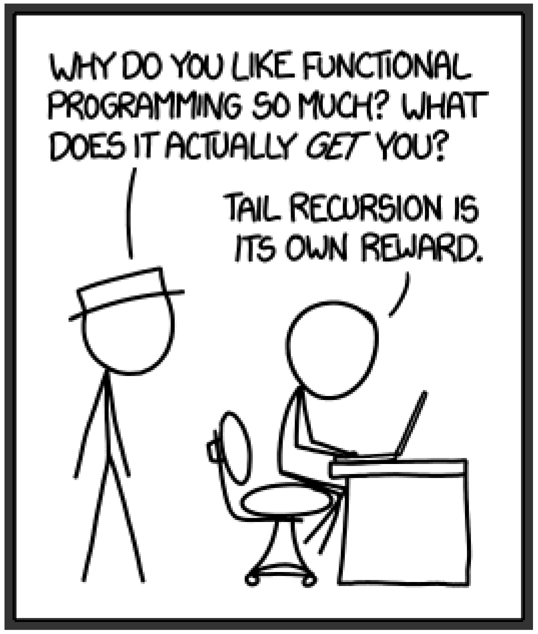
CHALLENGES?

- Designer Support
- After C#
- Before C#

CHALLENGES?

- People are too puritanical about purity.
  - be pragmatic and solution focused
  - functional purity not for the sake of itself

CHALLENGES?


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

XKCD

CHALLENGES?

- Explicit is better than implicit.
- Simple is better than Complex. Complex is better than Complicated.
- Special cases aren't special enough to break the rules. Although practicality beats purity.
- If the implementation is hard to explain, it's a bad idea.

---

the Zen of Python

```fsharp
let memoize f =
  let cache = System.Collections.Generic.Dictionary<_, _>()
  fun x ->
    if cache.ContainsKey(x) then cache.[x]
    else let res = f x
         cache.[x] <- res
         res

let somePureLongRunningFunc i =
  System.Threading.Thread.Sleep (2000)
  i + 1

let fastFunc = memoize somePureLongRunningFunc
```

CHALLENGE YOURSELF!

- "Practice does not make perfect. Only perfect practice makes perfect"
- "Perfection is not attainable. But if we chase perfection, we can catch excellence"

Vince Lombardi

CHALLENGE YOURSELF!

- "Programming languages have a devious influence: They shape our thinking habits."

  Edsger W. Dijkstra

- "One of the most disastrous thing we can learn is the first programming language, even if it's a good programming language."

  Alan Kay

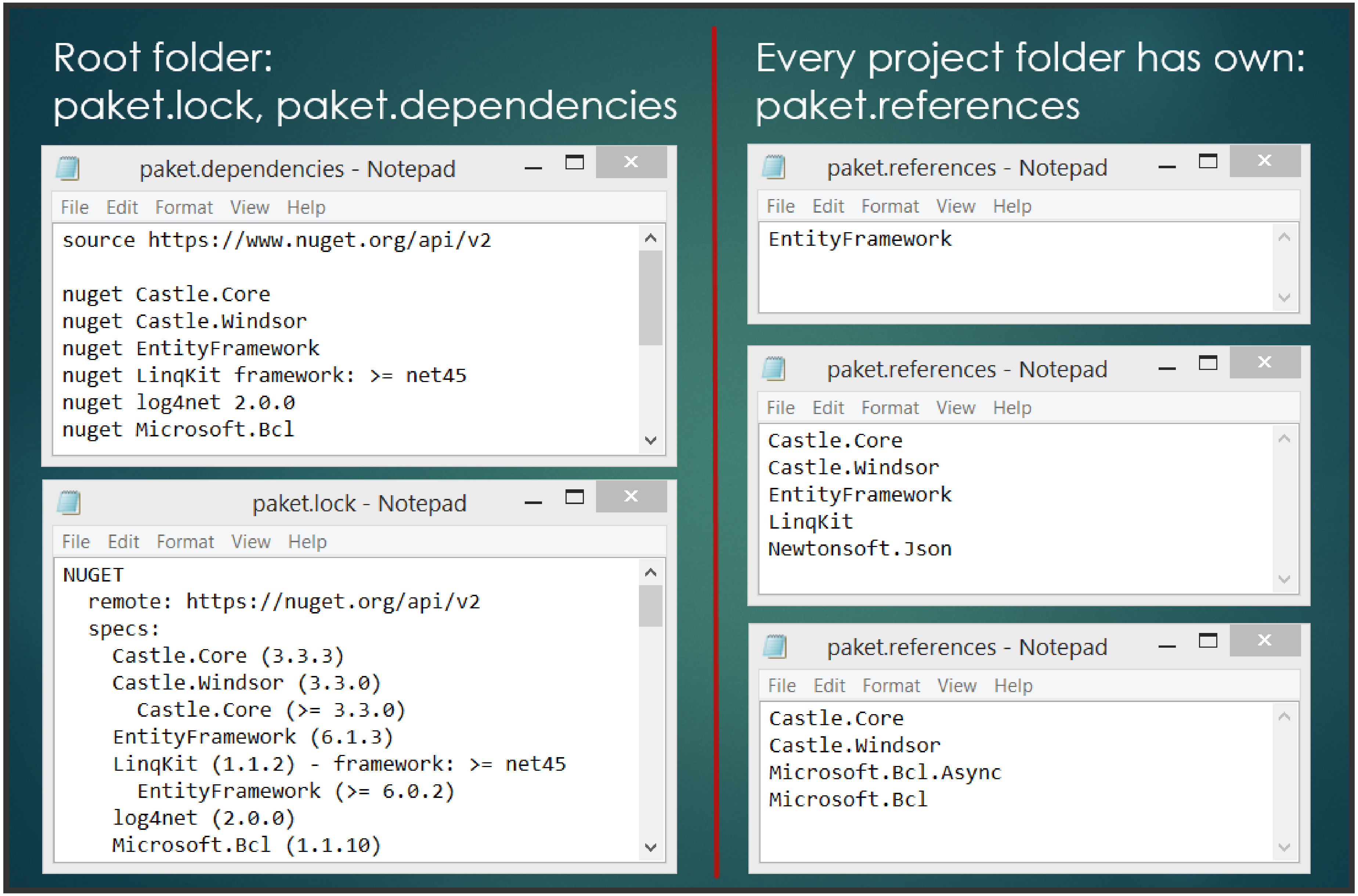
THANKS (AND THINGS TO READ/WATCH IN DEPTH)

- 7 ineffective coding habits many F# programmers don't have
- Does the language make a difference
- FSReveal (but really the whole F# community)
- Real world functional programming
- Luca Bolognese
- Enterprise F#

BONUS: FAKE

DEMO (these slides)

BONUS: PAKET



Root folder:
paket.lock, paket.dependencies

Every project folder has own:
paket.references

paket.dependencies - Notepad

```
source https://www.nuget.org/api/v2

nuget Castle.Core
nuget Castle.Windsor
nuget EntityFramework
nuget LinqKit framework: >= net45
nuget log4net 2.0.0
nuget Microsoft.Bcl
```

paket.lock - Notepad

```
NUGET
  remote: https://nuget.org/api/v2
  specs:
    Castle.Core (3.3.3)
    Castle.Windsor (3.3.0)
      Castle.Core (>= 3.3.0)
    EntityFramework (6.1.3)
    LinqKit (1.1.2) - framework: >= net45
      EntityFramework (>= 6.0.2)
    log4net (2.0.0)
    Microsoft.Bcl (1.1.10)
```

paket.references - Notepad

```
EntityFramework
```

paket.references - Notepad

```
Castle.Core
Castle.Windsor
EntityFramework
LinqKit
Newtonsoft.Json
```

paket.references - Notepad

```
Castle.Core
Castle.Windsor
Microsoft.Bcl.Async
Microsoft.Bcl
```

BONUS: PAKET

- Not a simple NuGet.exe replacement
- Package via csproj/fsproj (no nuspec needed)
- GitHub and Http (file based) dependencies.
- Caching
- Git dependencies (repository -> temporary replace nuget dependency)
- Support for most stuff (Roslyn Analyzers, Content files, ...)
- Doesn't support PowerShell scripts (by design)