In which I ridiculously over-engineer a simple game to create a real-world "enterprise-ready" application.

Enterprise Tic-Tac-Toe

Proper name is "Noughts and Crosses" btw

@ScottWlaschin
fsharpforfunandprofit.com

API design!

Type driven design!

Enterprise Tic-Tac-Toe

POLA &
Capability-based security

Parametric Polymorphism!

HATEOAS!

What you need for "Enterprise"?

- ~~Siloed organization~~ Specialized teams
- Architecture Team
- Project Management Team
- Front End Development Team
- Back End Development Team
- Operations Team
- Security Team
- Compliance and Auditing Team

Can we make them all happy?

What you need for "Enterprise"?

- **Separation of concerns** so that specialist teams can work on different parts of the code at the same time.
- **A documented API** so that the different teams can work effectively in parallel.
- **A security model** to prevent unauthorized actions from occurring.
- **Well-documented design** so that the architect can ensure that the implementation matches the UML diagrams.
- **Auditing and logging** to ensure that the system is compliant.
- **Scalability** to ensure that the system is ready for the challenges of rapid customer acquisition.

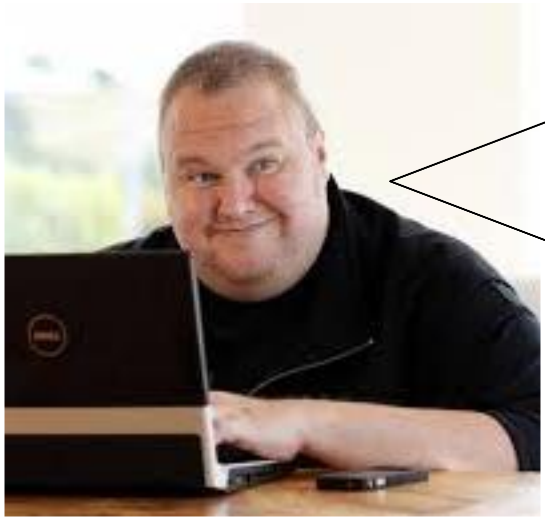
~~**Separation of concerns** so that specialist teams can work on different parts of the code at the same time.~~

Project Manager: "We need separation of concerns because the front-end team and back-end team hate each other and refuse to work in the same room."

~~A documented API so that the different teams can work effectively in parallel.~~

Front-end team: "We need a documented API so that those dummies building the back-end won't keep breaking our code on every commit."

~~A security model to prevent unauthorized actions from occurring.~~



Back-end team: "We need a security model because those idiots building the front-end will always find a way to do something stupid unless we constrain them."

~~**Well-documented design** so that the architect can ensure that the implementation matches the UML diagrams.~~

Maintenance team: "We need well-documented design because we're fed up of having to reverse engineer the hacked-up spaghetti being thrown at us."

~~**Auditing and logging** to ensure that the system is compliant.~~

Testers and Operations: "We need auditing and logging so that we can see what the effing system is doing inside."

~~Scalability to ensure that the system is ready for the challenges of rapid customer acquisition.~~

Everyone: "We don't really need scalability at all, but the CTO wants to us to be buzzword compliant."

I’m gonna use F#, so here’s all the F# you need

```
type Pair = int * int
```

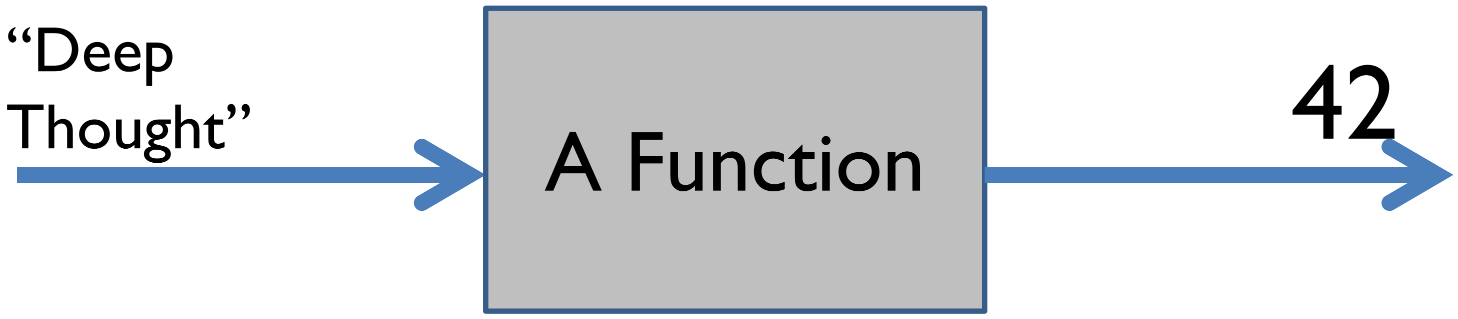
"*" means a pair

```
type Thing = { name:string }
```

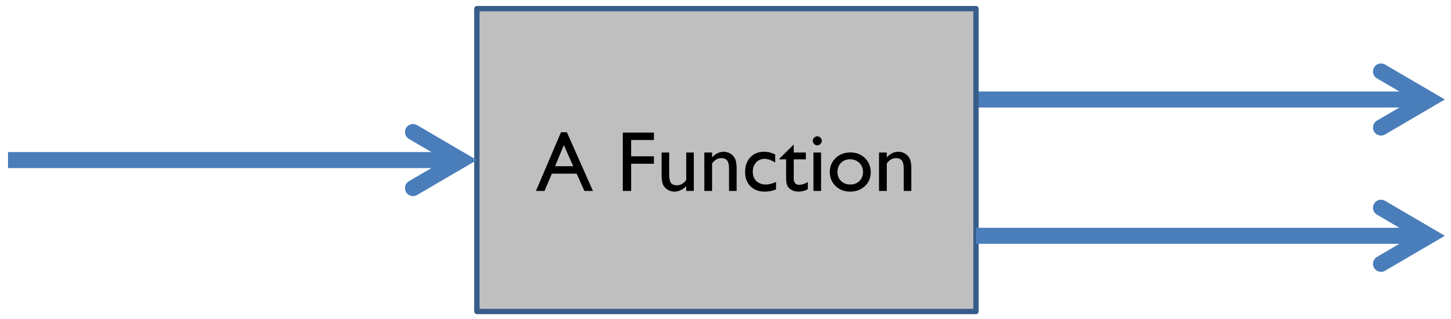
"{" means a record

```
type Payment =
| Cash
| Card of CardNumber
```

"|" means a choice

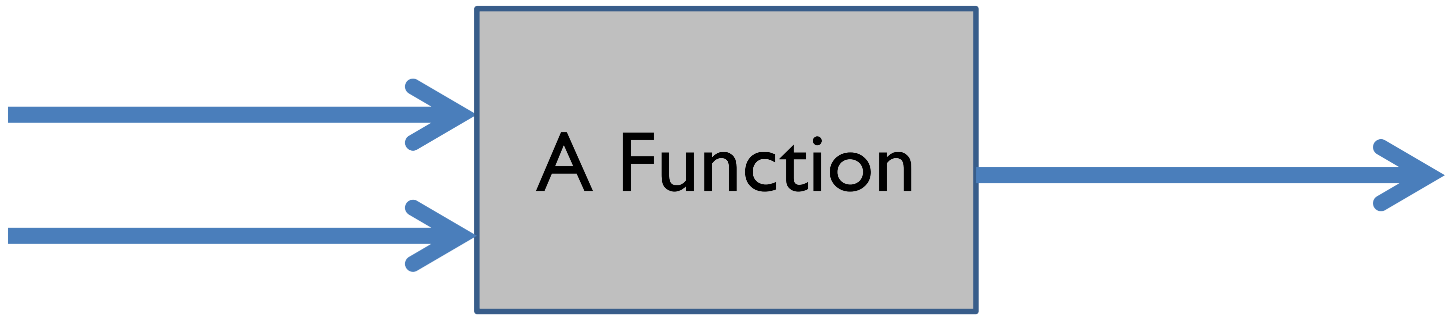


```
type AFunction =
    string ->
    int
```



```
type AFunction =
    string ->
    int * bool
```

Input is a string
Output is a pair



type AFunction =
  bool * string ->
  int

Input is a pair
Output is a int

TYPE DRIVEN DESIGN

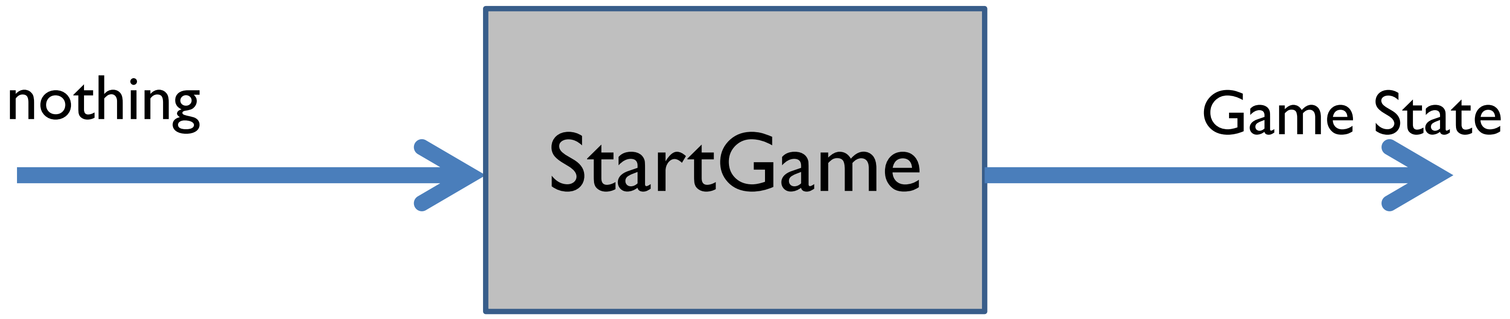
Growing functional software, guided by types

Type driven design

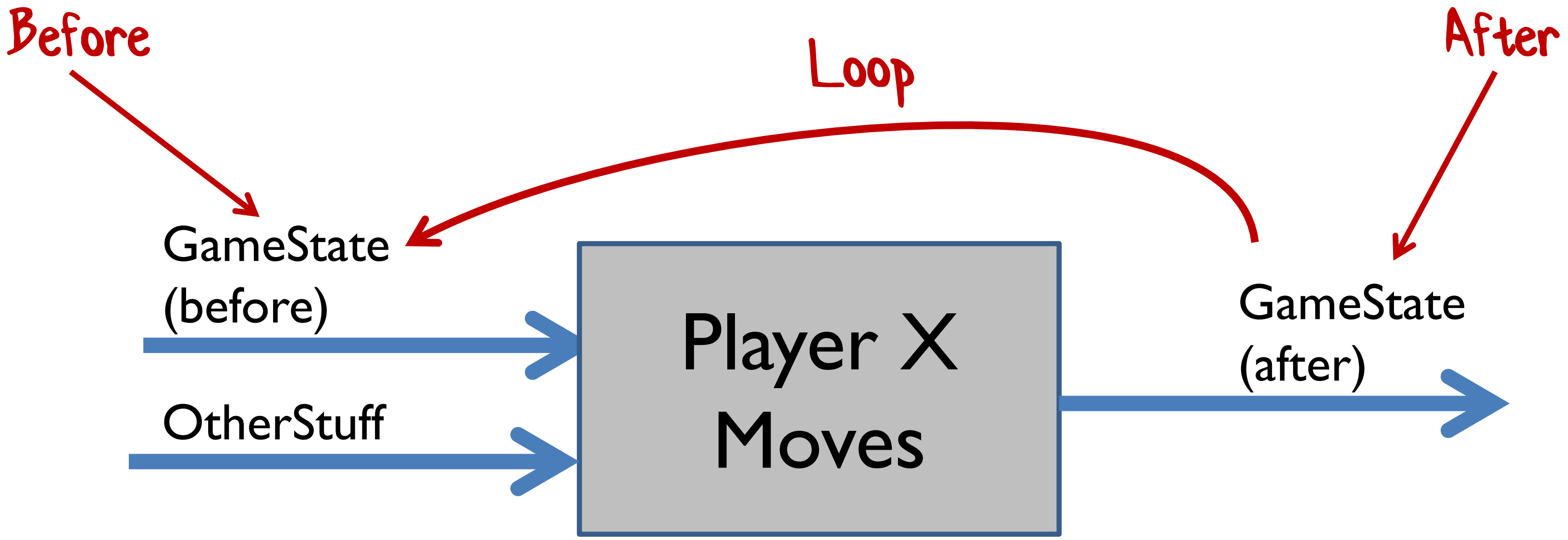
- Design with types only
  - no implementation code.
- Every use-case/scenario corresponds to a function type
  - one input and one output
- Work mostly top-down and outside-in
  - Occasionally bottom up as well.
- We ignore the UI for now.

Tic-Tac-Toe Scenarios

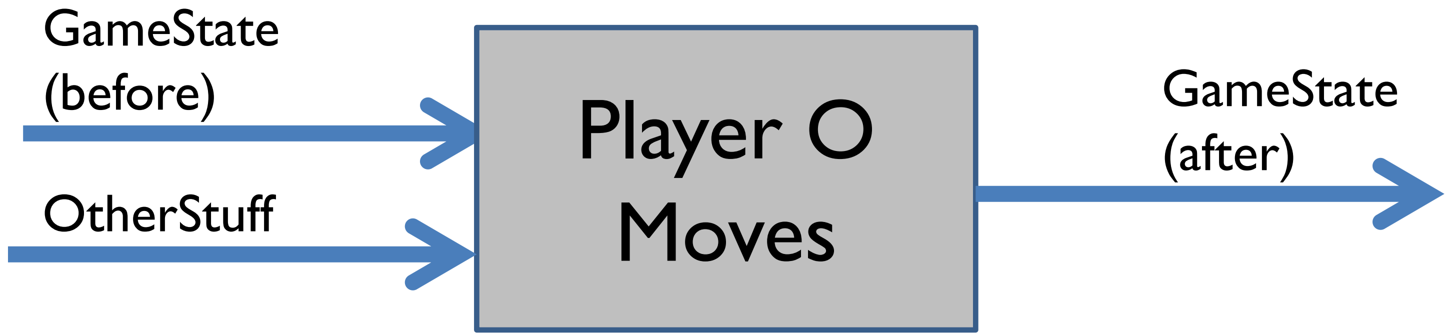
- Initialize a game
- A move by Player X
- A move by Player O



```
type StartGame =
    unit ->
    GameState
```

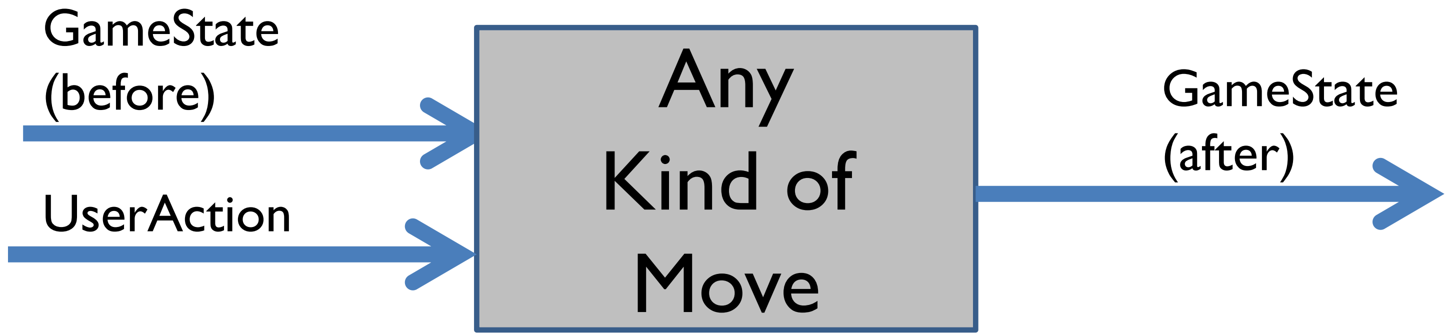


```
type PlayerXMove =
    GameState * SomeOtherStuff ->
    GameState
```



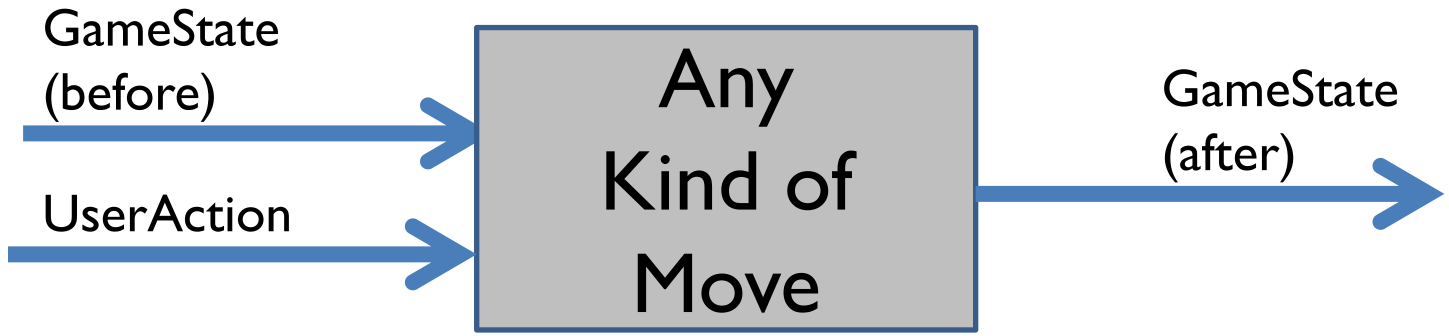
```
type PlayerOMove =
    GameState * SomeOtherStuff ->
    GameState
```

☹ both functions look exactly the same and could be easily substituted for each other.



```
type UserAction =
   | MoveLeft
   | MoveRight
   | Jump
   | Fire
```

Generic approach



```
type UserAction =
   | PlayerXMove of SomeStuff
   | PlayerOMove of SomeStuff
```

Generic approach applied to this game

But we have TWO players so should have two functions....

```
type PlayerXMove =
    GameState * PlayerX's Stuff ->
    GameState

type PlayerOMove =
    GameState * PlayerO's Stuff ->
    GameState
```

Each type is different and the compiler won't let them be mixed up!

What is the other Stuff?

For some domains there might be a LOT of stuff...
But in Tic-Tac-Toe, it's just the location on the grid where the player makes their mark.

```
type HorizPosition =
   Left | Middle | Right

type VertPosition =
   Top | Center | Bottom

type CellPosition =
   HorizPosition * VertPosition
```

```
type PlayerXMove =
    GameState * CellPosition ->
    GameState

type PlayerOMove =
    GameState * CellPosition ->
    GameState
```

Same again ☹

```
type PlayerXPos =
    PlayerXPos of CellPosition

type PlayerOPos =
    PlayerOPos of CellPosition
```

Different positions

```
type PlayerXMove =
    GameState * PlayerXPos ->
    GameState


type PlayerOMove =
    GameState * PlayerOPos ->
    GameState
```

Different functions☺

What is the GameState?

```
type GameState = { cells : Cell list }
```

```
type CellState =
  | X
  | O
  | Empty
```

```
type Cell = {
   pos : CellPosition
   state : CellState }
```

What is the GameState?

```
type GameState = { cells : Cell list }
```

```
type Player = PlayerX | PlayerO

type CellState =
   | Played of Player
   | Empty
```

Refactor!

```
type Cell = {
   pos : CellPosition
   state : CellState }
```

What is the Output?

What does the UI need to know?

The UI should not have to "think" -- it should just follow instructions.

What is the Output?

1) Pass the entire game state to the UI?

But the GameState should be opaque...

What is the Output?

~~1) Pass the entire game state to the UI?~~

2) Make the UI's life easier by explicitly returning the cells that changed with each move

```
type PlayerXMove =
    GameState * PlayerXPos ->
    GameState * ChangedCells
```

Too much trouble in this case

What is the Output?

~~1) Pass the entire game state to the UI?~~

~~2) Make the UI's life easier by explicitly returning the cells that changed with each move~~

3) The UI keeps track itself but can ask the server if it ever gets out of sync

```
type GetCells = GameState -> Cell list
```

Time for a walkthrough...

Start game
Player X moves
Player O moves
Player X moves
Player O moves
Player X moves
Player X wins!

Time for a walkthrough...

Start game
Player X moves
Player O moves
Player X moves
Player O moves
Player X moves
Player X wins!
Player O moves
Player X moves
Player O moves
Player X moves
Player O moves
Player X moves
Player O moves
Player X moves
Player O moves

Did I mention that the UI was stupid?

When does the game stop?

How does the UI know?

```
type GameStatus =
   | InProcess
   | Won of Player
   | Tie
```

```
type PlayerXMove =
  GameState * PlayerXPos ->
  GameState * GameStatus
```

Returned with the GameState

Review

What kind of errors can happen?

- **Could the UI create an invalid GameState?**
  - No. We're going to keep the internals of the game state hidden from the UI. ✓
- **Could the UI pass in an invalid CellPosition?**
  - No. The horizontal/vertical parts of CellPosition are restricted. ✓
- **Could the UI pass in a valid CellPosition but at the wrong time?**
  - Yes -- that is totally possible. ✗
- **Could the UI allow player X to play twice in a row?**
  - Again, yes. Nothing in our design prevents this. ✗
- **What about when the game has ended but the stupid UI forgets to check the GameStatus and doesn't notice.**
  - The game logic needs to not accept moves after the end! ✗

Returning the available moves

```
type ValidMovesForPlayerX = PlayerXPos list
type ValidMovesForPlayerO = PlayerOPos list
```

```
type PlayerXMove =
    GameState * PlayerXPos ->
    GameState * GameStatus * ValidMovesForPlayerO

type PlayerOMove =
    GameState * PlayerOPos ->
    GameState * GameStatus * ValidMovesForPlayerX
```

Now returned after each move

What kind of errors can happen?

```
type PlayerXMove =
    GameState * PlayerXPos ->
    GameState * GameStatus * ValidMovesForPlayerO

type PlayerOMove =
    GameState * PlayerOPos ->
    GameState * GameStatus * ValidMovesForPlayerX
```

- **Could the UI pass in a valid CellPosition but at the wrong time?**
  - No, it is not in the list of allowed moves. ✓
- **Could the UI allow player X to play twice in a row?**
  - No, the returned list only has moves for Player O ✓
- **What about when the game has ended but the stupid UI forgets to check the GameStatus and doesn't notice.**
  - The list of moves is empty when the game is over ✓

Some refactoring (before)

```
type GameStatus =
   | InProcess
   | Won of Player
   | Tie

type PlayerXMove =
    GameState * PlayerXPos ->
    GameState * GameStatus * ValidMovesForPlayerO
```

Merge into one type

Some refactoring (after)

```
type MoveResult =
    | PlayerXToMove of GameState * ValidMovesForPlayerX
    | PlayerOToMove of GameState * ValidMovesForPlayerO
    | GameWon of GameState * Player
    | GameTied of GameState
```

```
type PlayerXMove =
    GameState * PlayerXPos -> MoveResult

type PlayerOMove =
    GameState * PlayerOPos -> MoveResult
```

Time for a demo!

Hiding implementations with Parametric Polymorphism

Hiding implementations with ~~Parametric Polymorphism~~ Generics

Enforcing encapsulation

- Decouple the "interface" from the "implementation".
- Shared data structures that are used by both the UI and the game engine. (CellState, MoveResult, PlayerXPos, etc.)
- Private data structures that should only be accessed by the game engine (e,g. GameState)

Enforcing encapsulation

- OO approaches:
  - Represent GameState with an abstract base class
  - Represent GameState with an interface
  - Make constructor private

Enforcing encapsulation

- FP approach:
  - Make the UI use a generic GameState
  - GameState can stay public
  - All access to GameState internals is via functions
    - These functions "injected" into the UI

With List<T>, you can work with the list in many ways, but you cannot know what the T is, and you can never accidentally write code that assumes that T is an int or a string or a bool.

This "hidden-ness" is not changed even when T is a public type.

With a generic GameState

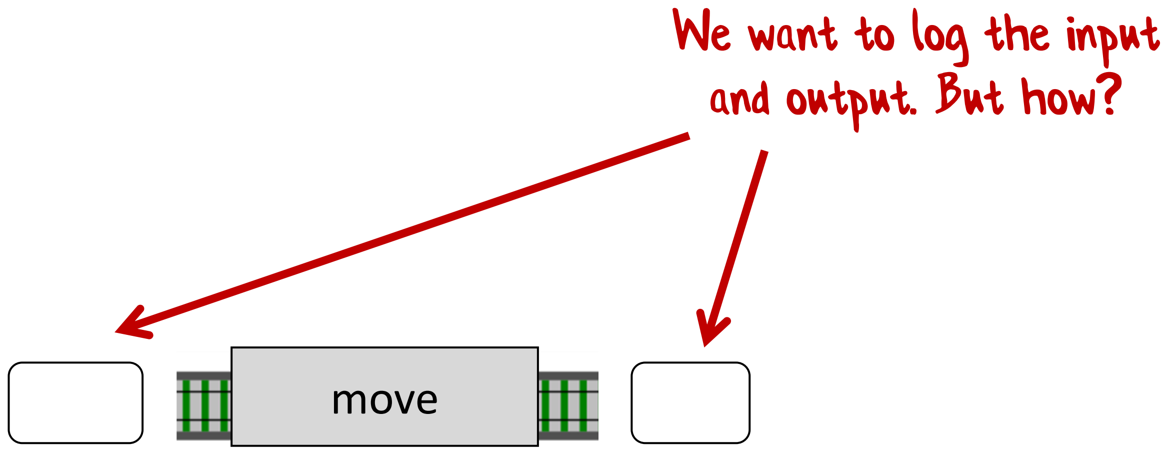
```
type PlayerXMove<'GameState> =
   'GameState * PlayerXPos ->
   'GameState * MoveResult

type PlayerOMove<'GameState> =
   'GameState * PlayerOPos ->
   'GameState * MoveResult
```

The UI is injected with these functions but doesn't know what the GameState *really* is.

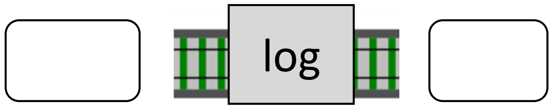
Logging

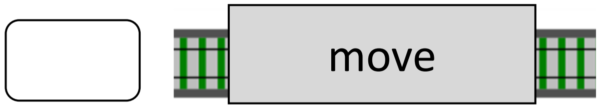
Logging



Logging

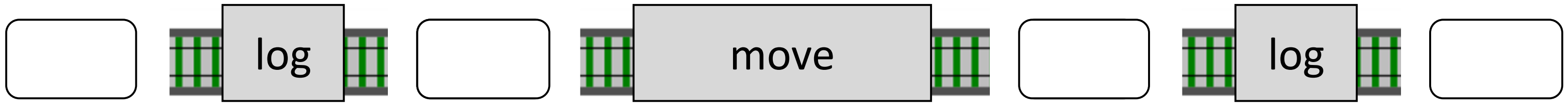
Step 1: Create a log function





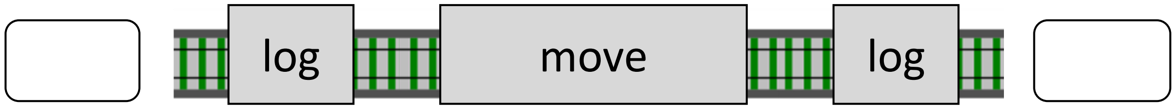
Logging

Step 2: glue all the functions together using composition



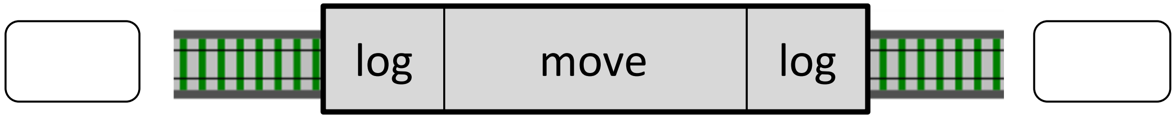
Logging

Step 2: glue all the functions together using composition



Logging

Step 3: use the new function in place of old function



There's no need for a "decorator pattern" in FP - it's just regular composition

Demo of logging

Client-server communication

How do you send domain objects on the wire?

What communication method should we use?

JSON over HTTP?

Enterprise Rating: C-

✗

Too hipster ☹

What communication method should we use?

XML & SOAP?

Enterprise Rating: A ✗

Good, but we can do better...

What communication method should we use?

Enterprise Service Bus!

Enterprise Rating: A++

Ultimate sign of enterprisey-ness

Sending objects on the wire

```
type MoveResult =
    | PlayerXToMove of GameState * ValidMovesForPlayerX
    | PlayerOToMove of GameState * ValidMovesForPlayerO
    | GameWon of GameState * Player
    | GameTied of GameState
```

Not serialization friendly

```
type MoveResultDTO = {
    moveResultType : string   // e.g. "PlayerXToMove"
    gameStateToken : string
    // only applicable in some cases
    availableMoves : int list
    }
```

JSON/XML friendly

Demo of problems

Stupid people

Evil people

What's the difference? ☹

POLA & Capability Based Security

Evolution of a configuration API

Say that the UI needs to set a configuration option (e.g. DontShowThisMessageAgain)

How can we stop a malicious caller doing bad things?

Attempt 1
Give the caller the configuration file name

API

```
interface IConfiguration
{
    string GetConfigFilename();
}
```

Caller

```
var filename = config.GetConfigFilename();
// open file
// write new config
// close file
```

☹ A malicious caller has the ability to write to any file on the filesystem

Attempt 2
Give the caller a TextWriter

API

```
interface IConfiguration
{
    TextWriter GetConfigWriter();
}
```

Caller

```
var writer = config.GetConfigWriter();
// write new config
```

☹ A malicious caller can corrupt the config file

Attempt 3
Give the caller a key/value interface

API

```
interface IConfiguration
{
  void SetConfig(string key, string value);
}
```

Caller

```
config.SetConfig(
  "DontShowThisMessageAgain", "True");
```

☹ A malicious caller can set the value to a non-boolean

Attempt 4
Give the caller a domain-centric interface

API

```
enum MessageFlag {
   ShowThisMessageAgain,
   DontShowThisMessageAgain
   }

interface IConfiguration
{
    void SetMessageFlag(MessageFlag value);
    void SetConnectionString(ConnectionString value);
    void SetBackgroundColor(Color value);
}
```

☹ What's to stop a malicious caller changing the connection string when they were only supposed to set the flag?

Attempt 5
Give the caller only the interface they need

API

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

☺ The caller can *only* do the thing we allow them to do.

Good security implies good design

Good security is good design

- Filename => limit ourselves to file-based config files.
  - A TextWriter makes the design is more mockable
- TextWriter => exposing a specific storage format
  - A generic KeyValue store make implementation choices more flexible.
- KeyValue store using strings means possible bugs
  - Need to write validation and tests for that ☹
  - Statically typed interface means no corruption checking code. ☺
- An interface with too many methods means no ISP
  - Reduce the number of available methods to one!

Capability based design

- In a cap-based design, the caller can only do exactly one thing -- a "capability".
- In this example, the caller has a capability to set the message flag, and that's all.

Stops malicious AND stupid callers doing bad things!

Attempt 5
A one method interface is a function

OO API

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

Functional API

```
Action<MessageFlag> messageFlagCapability
```

Capability Based Security and Tic-Tac-Toe

Switching to cap-based Tic-Tac-Toe

```
type MoveResult =
   | PlayerXToMove of
         DisplayInfo * NextMoveInfo list
   | PlayerOToMove of
         DisplayInfo * NextMoveInfo list
   | GameWon of DisplayInfo * Player
   | GameTied of DisplayInfo
```

```
type NextMoveInfo = {
  posToPlay : CellPosition
  capability : MoveCapability }
```

This is for UI information only. The position is "baked" into the capability

This is a function

Cap-based Demo

RESTful done right

HATEOAS
Hypermedia As The Engine Of Application State

"A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia."

How NOT to do HATEOAS

```
POST /customers/
GET /customer/42
```

If you know the API
you're doing it wrong

How to do HATEOAS

```
POST /81f2300b618137d21d /
GET /da3f93e69b98
```

You can only know what URLs to use by parsing the page

HATEOAS Demo

Some Benefits of HATEOAS

- The server owns the API model and can change it without breaking any clients
  - E.g. Change links to point to CDN
  - E.g. Versioning
- Simple client logic
- Explorable API

Review: How “enterprise” are we?

- Separation of concerns ✓
- A documented API ✓
- Well-documented design ✓

```
type MoveResult =
    | PlayerXToMove of
          GameState * ValidMovesForPlayerX
    | PlayerOToMove of
          GameState * ValidMovesForPlayerO
    | GameWon of GameState * Player
    | GameTied of GameState
```

Review: How "enterprise" are we?

- A security model ✓
- Auditing and logging ✓
- Scalability ✗

You can just waffle here:
"immutable" blah blah blah
"no side effects" blah blah blah
Your CTO will be impressed.

Thanks!

fsharp.org

Contact me

@ScottWlaschin

fsharpforfunandprofit.com/ettt

Slides and video here

fsharpworks.com

Let us know if you need help with F#

More F# at fsharp.org