

# Deep dive into the design of TicTacToe

With a look at "capabilities" and  
security.

# Type driven approach

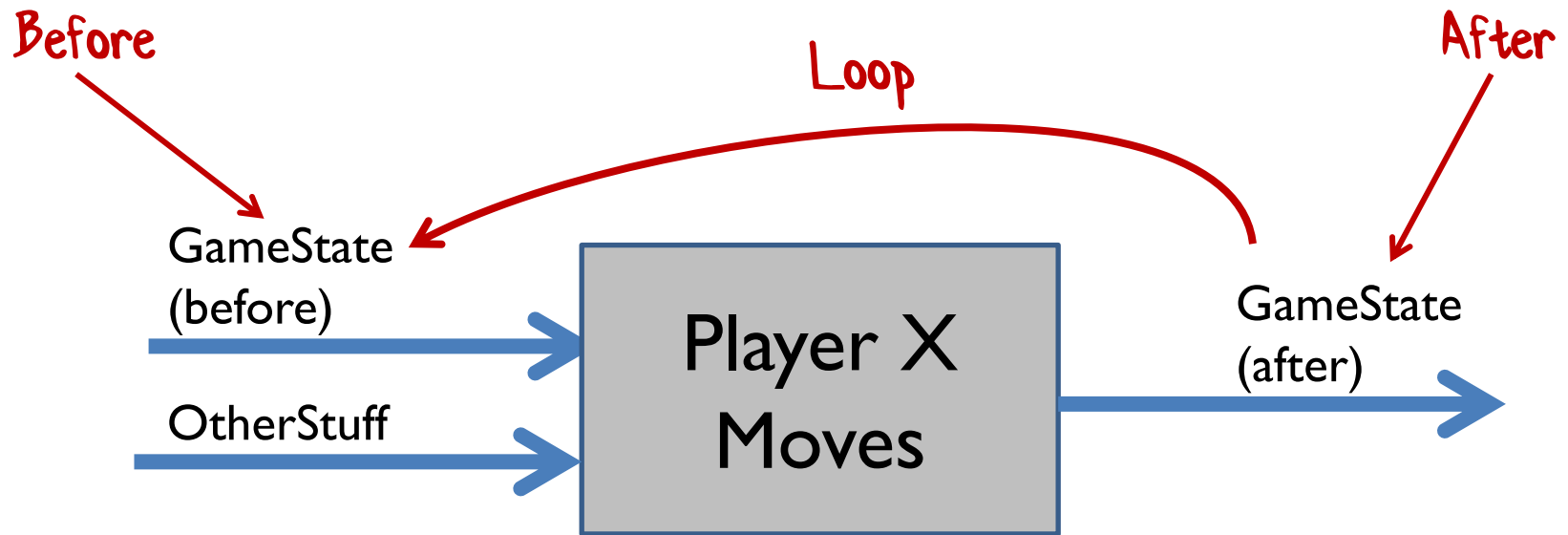
- Design with types only
  - no implementation code.
- Every use-case/scenario corresponds to a function type
  - one input and one output
- Work mostly top-down and outside-in
  - Occasionally bottom up as well.
- We ignore the UI for now.

# 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*



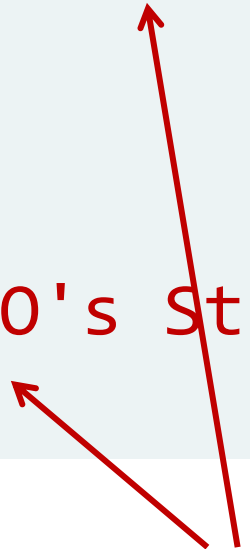
```
type UserAction =  
  | PlayerXMove of SomeStuff  
  | PlayerOMove of SomeStuff
```

*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 😞



All problems can be solved by wrapping them in a type!

```
type PlayerXPos =  
    PlayerXPos of CellPosition
```

Different  
positions

```
type PlayerOPos =  
    PlayerOPos of CellPosition
```

```
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 GameState =  
  | InProcess  
  | Won of Player  
  | Tie
```






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

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! 

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

*Downside: you have to change the GameState type to support these approaches*

# 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.

# Client-server communication

How do you send domain  
objects on the wire?

# 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  
}
```

Every field is a primitive



JSON/XML friendly

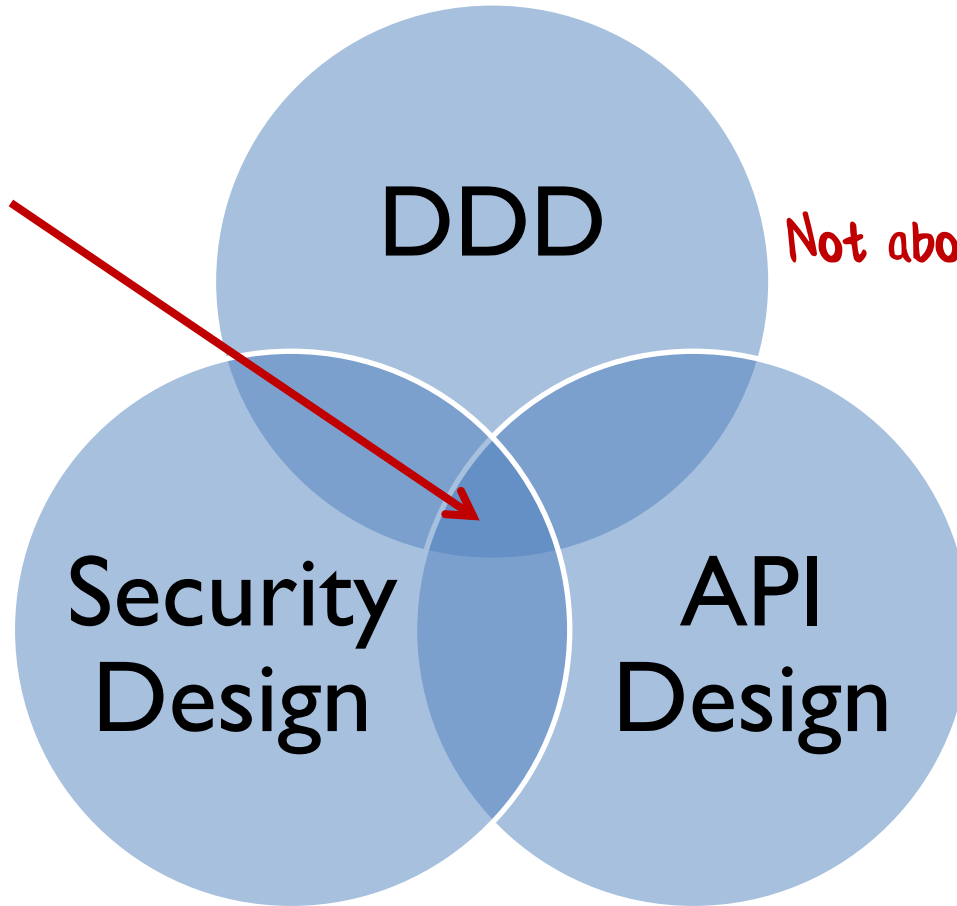




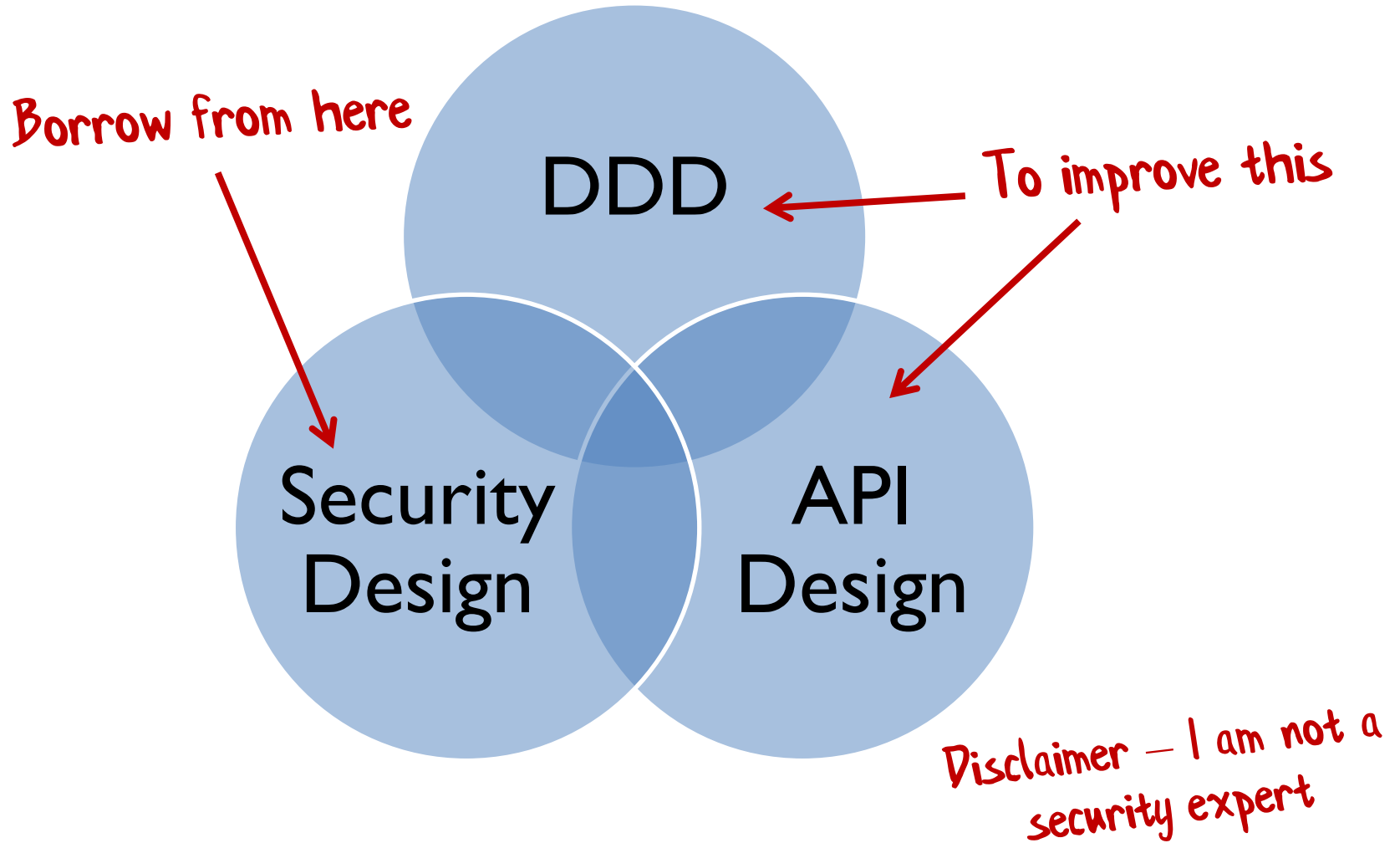
# Demo of problems

# **Designing with capabilities**

*This bit*



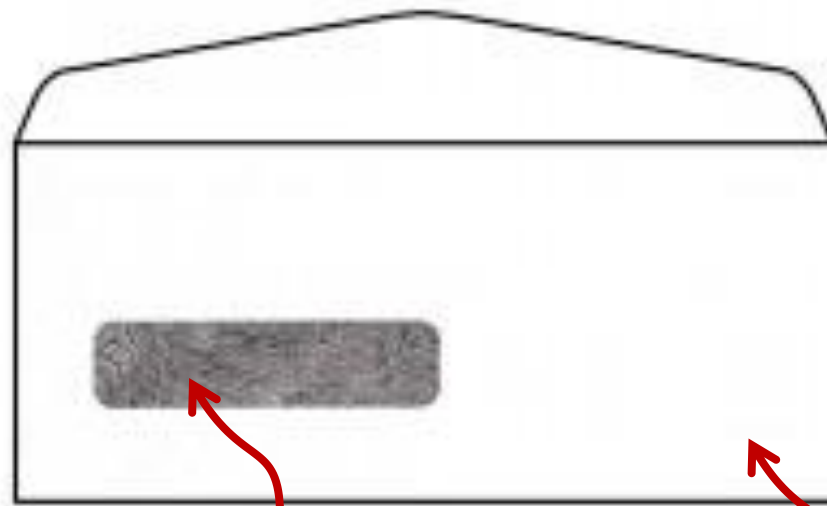
*Not about OAuth, JWT etc*



# Topics

- What does security have to do with design?
- Introducing capabilities
- API design with capabilities
- Design consequences of using capabilities
- Transforming capabilities for business rules
- Delegating authority using capabilities

**WHAT DOES SECURITY  
HAVE TO DO WITH DESIGN?**



Transparent

Opaque

It's all about  
security, right?

Please deliver  
this letter



Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, Temporibus autem quibus Dacei Megasystems Tech Inc necessitatibus aut officiis debitis aucto 2799 E Dragam Suite 7 quisquam saepe itaque enieti Los Angeles CA 90002 ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?

A counterexample



Please deliver  
this letter

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, Temporibus autem quibus Dace Megasystems Tech Inc necessitatibus aut officiis debitis auteo 2799 E Dragam Suite 7 quisquam saepe itaque eniet Los Angeles CA 90002 ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?

It's not just  
about security...

...hiding irrelevant  
information is  
good design!

# David Parnas, 1971

- If you make information available:
  - Programmers can't help but make use of it
  - Even if not in best interests of the design
- Solution:
  - Don't make information available!

# Software Design Spectrum

In the large: Bounded Contexts  
In the small: Interface Segregation Principle



Can't do  
anything

**Just right**

Unnecessary  
coupling



*Too little information available*

*Too much information available*

# Security spectrum

Principle of Least Authority (POLA)



Can't get your  
work done

**Just right**

Potential for  
abuse



*Too little information available*

*Too much information available*

# Good Software Design

Intention-revealing interface

Minimize coupling

Make dependencies explicit

*Ak.a. Minimize your surface area  
(expose only desired behavior)*

## Good Security

Principle of Least Authority (POLA)

*Ak.a. Minimize your surface area  
(to reduce chance of abuse)*

Good security  $\Rightarrow$  Good design

Good design  $\Rightarrow$  Good security

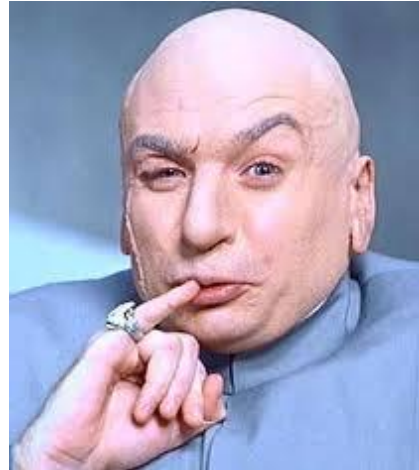
# Security-aware design

- "Authority" = what can you do at any point?
  - Be aware of authority granted
  - Assume malicious users as a design aid!

Stupid people



Evil people



What's the difference? ☹️

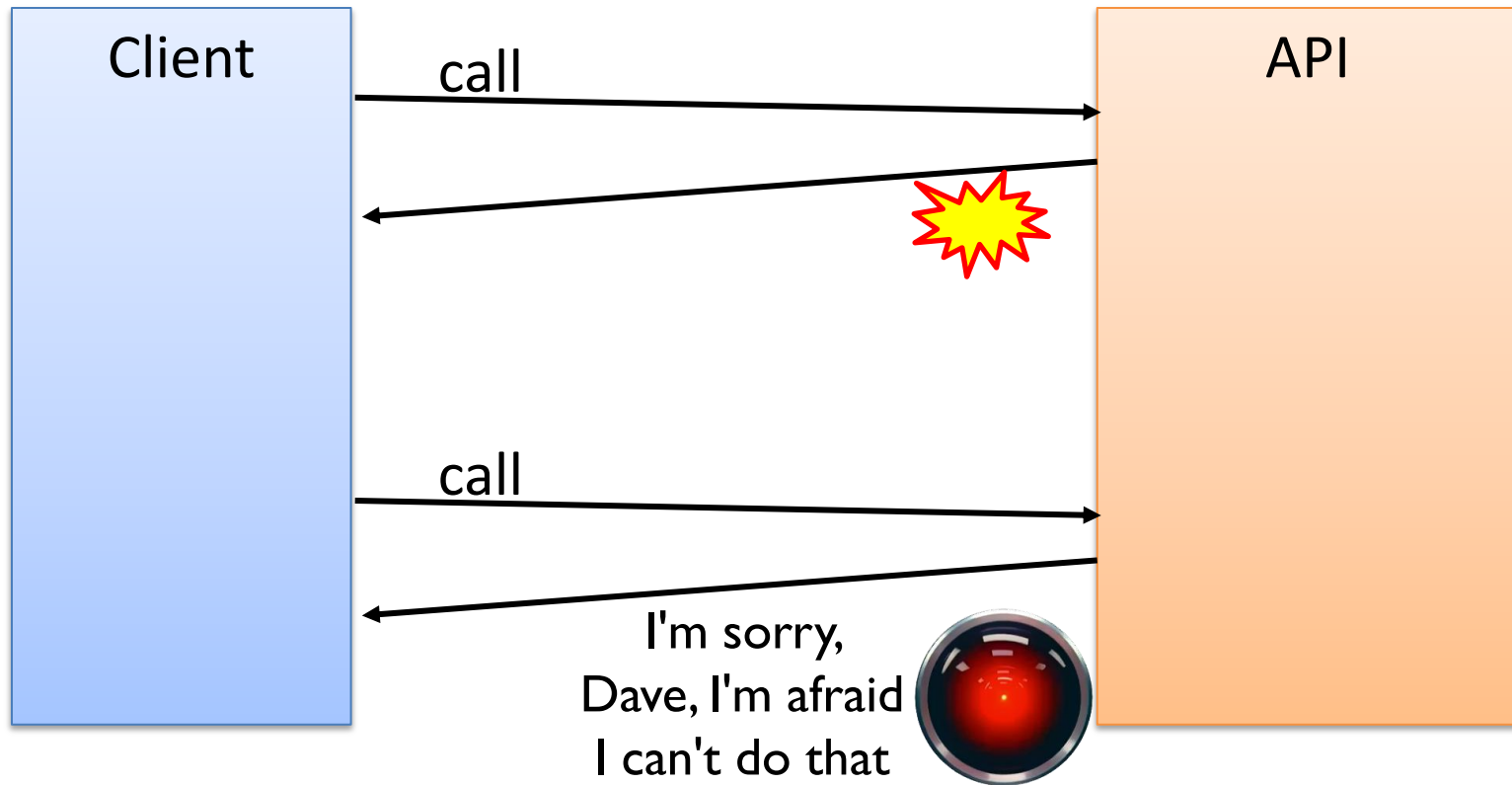


# Security-aware design

- "Authority" = what can you do at any point?
  - Be aware of authority granted
  - Assume malicious users as a design aid!
- Use POLA as a software design guideline
  - Forces intention-revealing interface
  - Minimizes surface area & reduces coupling

# **INTRODUCING “CAPABILITIES”**

# Typical API



Rather than telling me what I **can't** do,  
why not tell me what I **can** do?

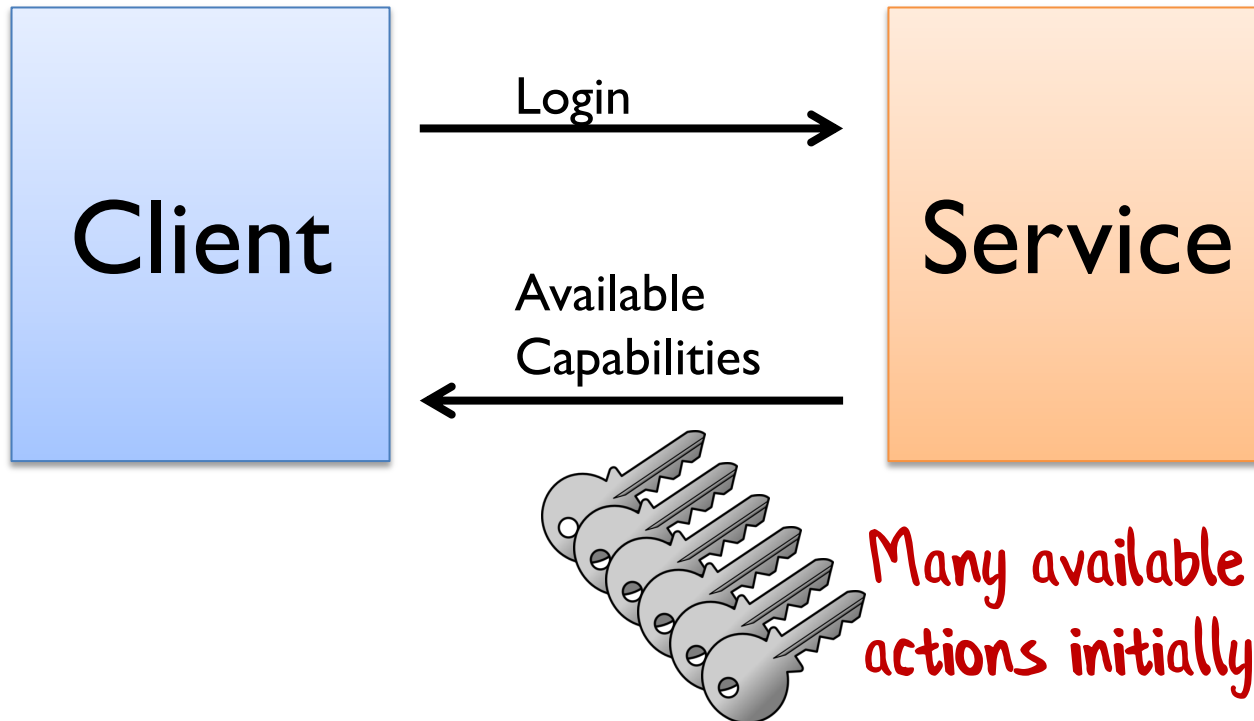
The ultimate  
"Intention-revealing interface"

# Capability-based API

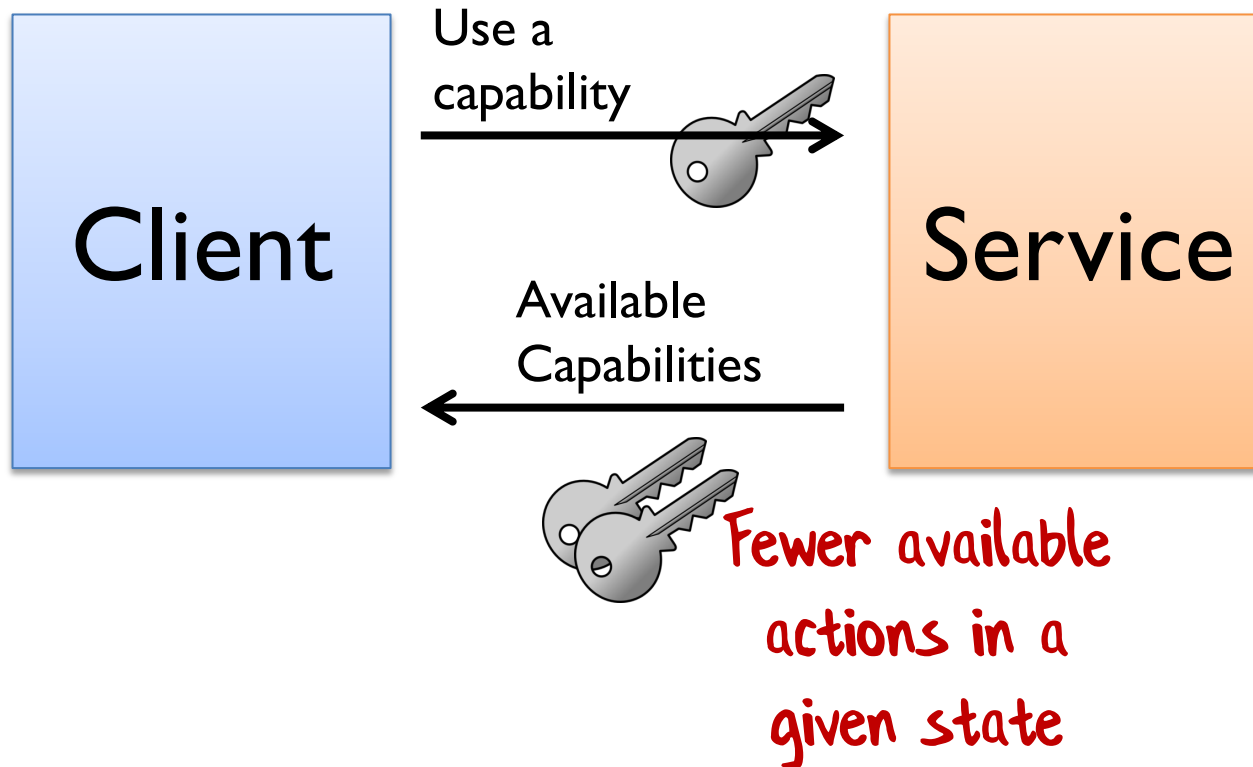


*A capability*

# Capability-based API



# Capability-based API



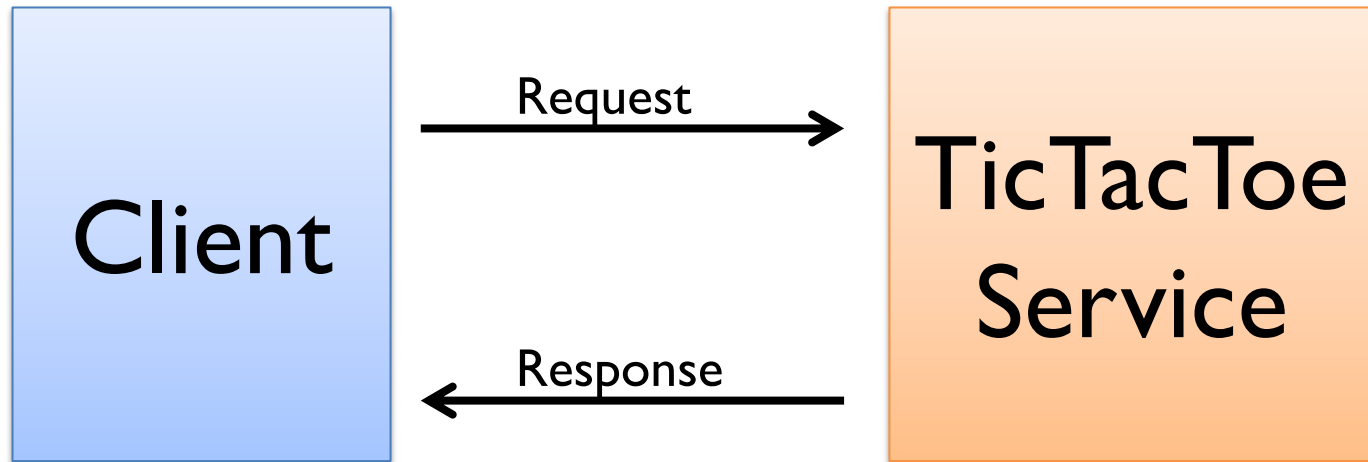
O		X
X	X	O
O		

# API DESIGN WITH CAPABILITIES



# Tic-Tac-Toe as a service

Proper name is "Noughts and Crosses" btw  
TIL: "Butter, cheese and eggs" in Dutch



# Tic-Tac-Toe API (obvious version)

```
type TicTacToeRequest = {  
  player: Player // X or O  
  row: Row  
  col: Column  
}
```

# Tic-Tac-Toe API (obvious version)

```
type TicTacToeResponse =  
  | KeepPlaying  
  | GameWon of Player  
  | GameTied
```

# Demo:

## Obvious Tic-Tac-Toe API

# What kind of errors can happen?

- A player can play an already played move
- A player can play twice in a row
- A player can forget to check the response and keep playing

*Not an intention-revealing interface*

# Intention-revealing interface

*"If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost."*

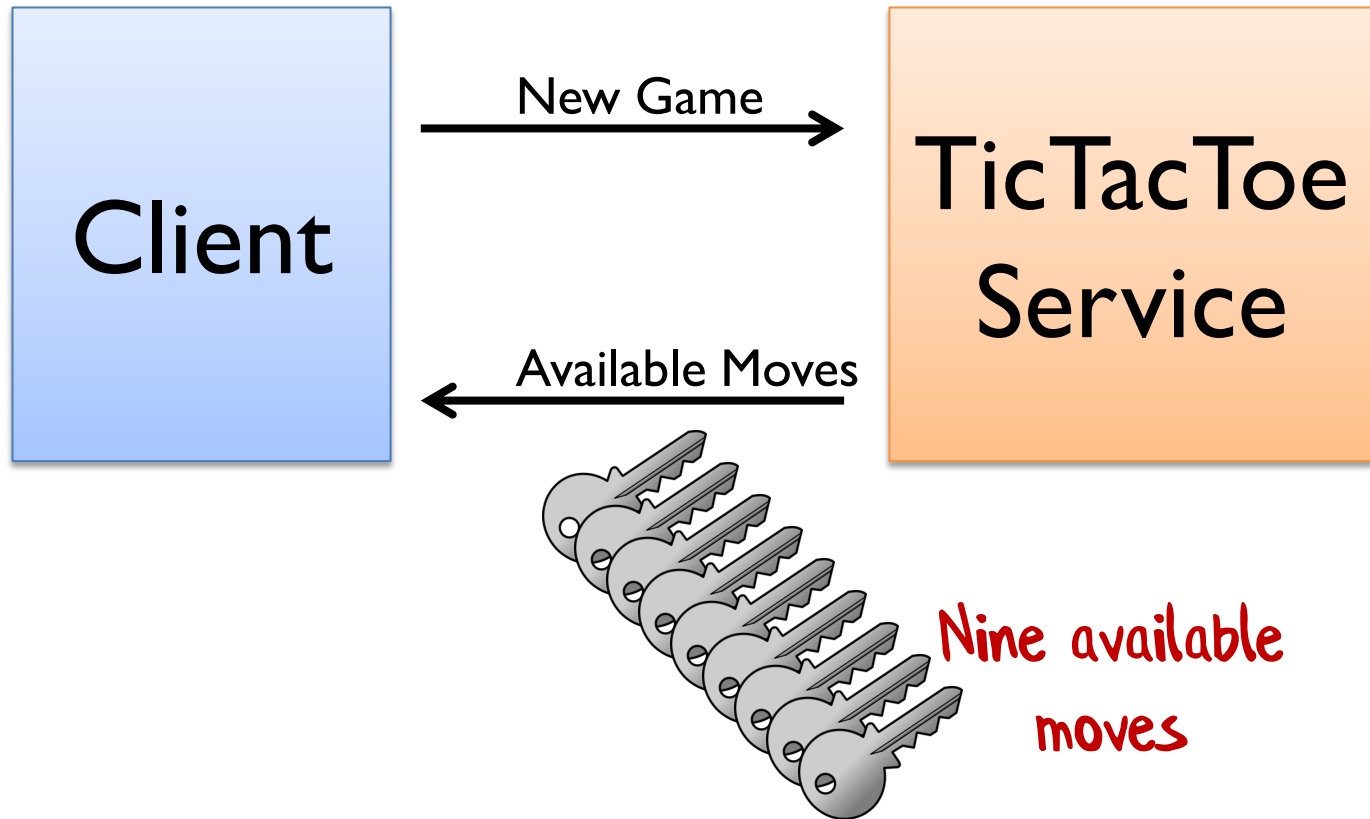
*-- Eric Evans, DDD book*

Yes, you could return errors, but...

Don't let me do a bad thing and  
then tell me off for doing it...

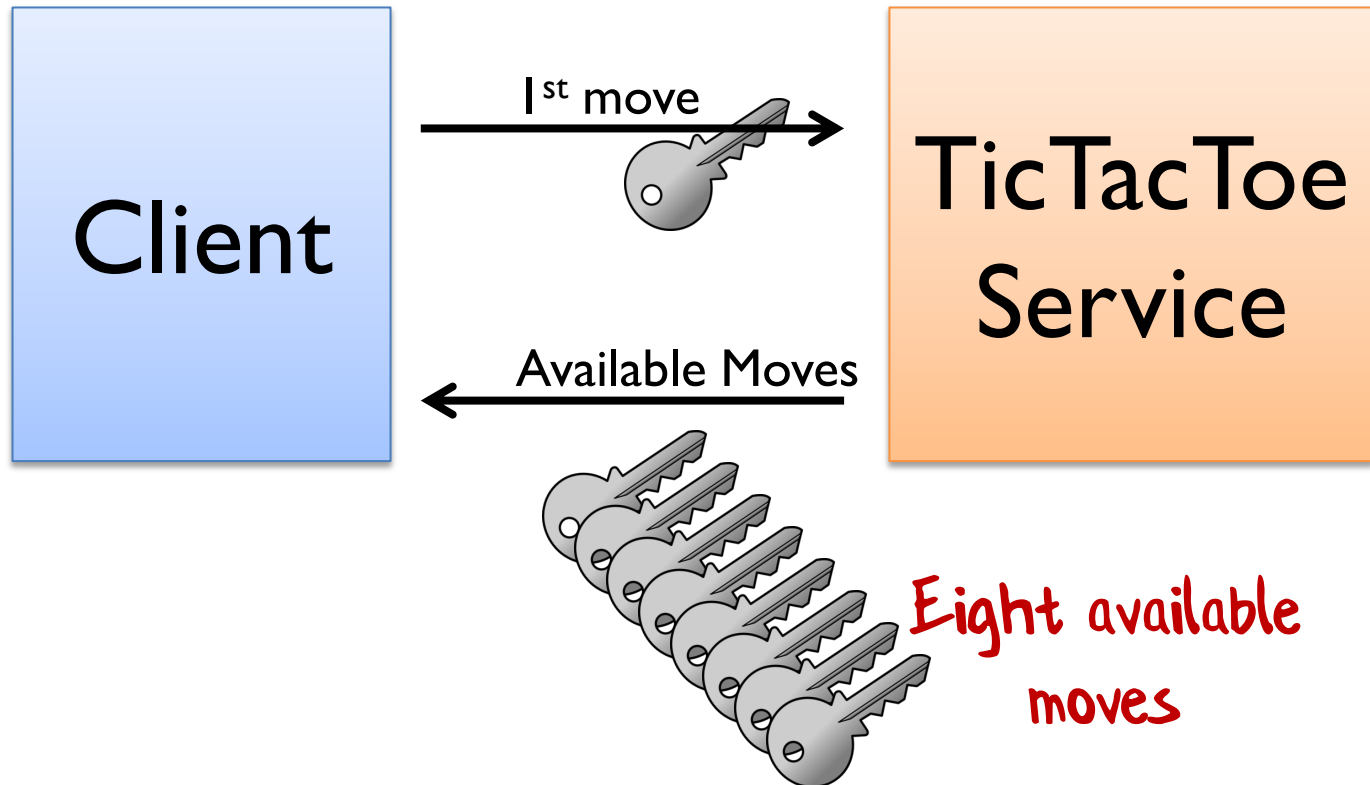
**“Make illegal operations  
unavailable”**

# Tic-Tac-Toe service with capabilities

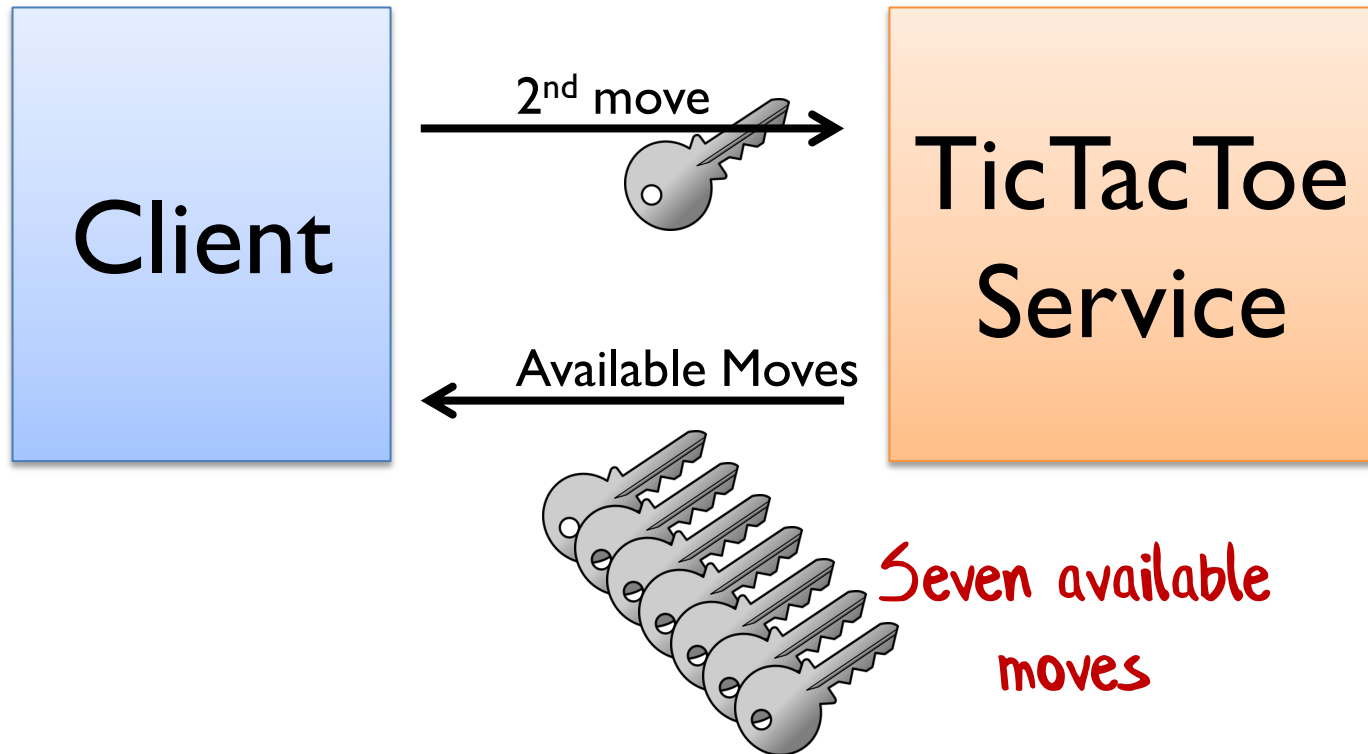




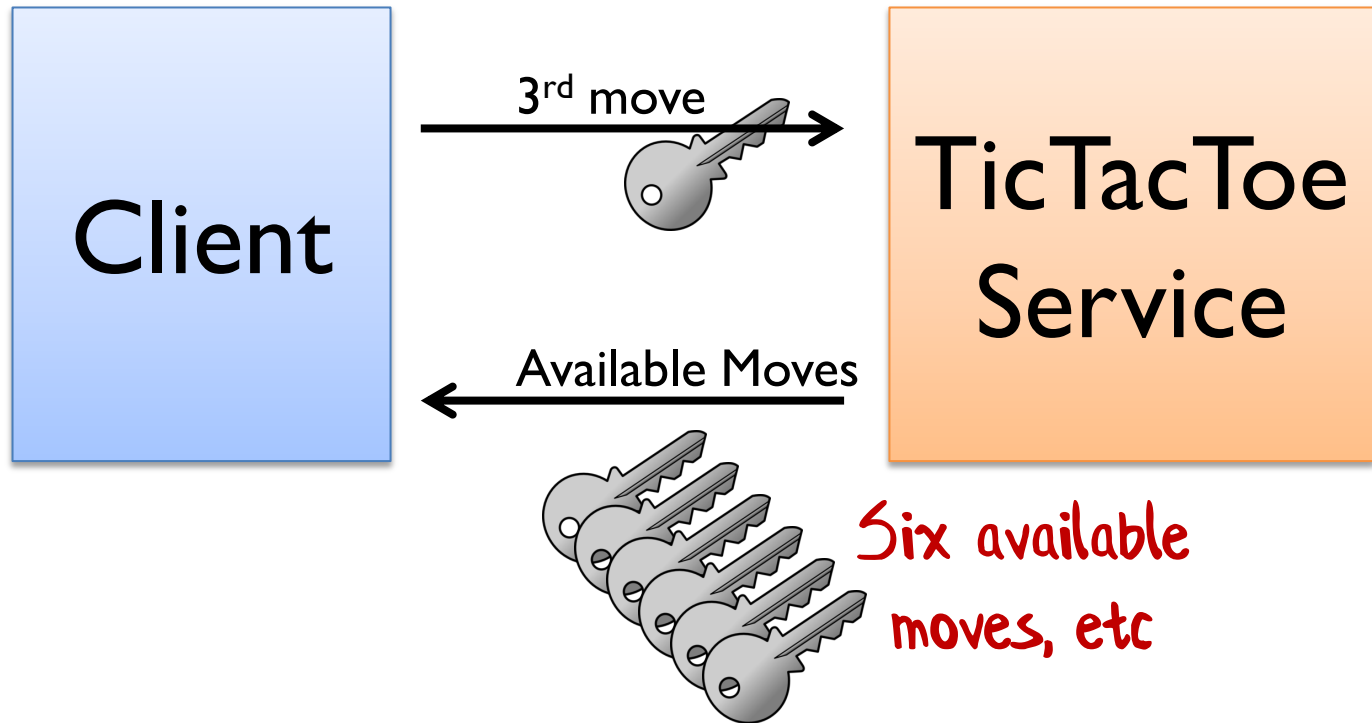
# Tic-Tac-Toe service with capabilities



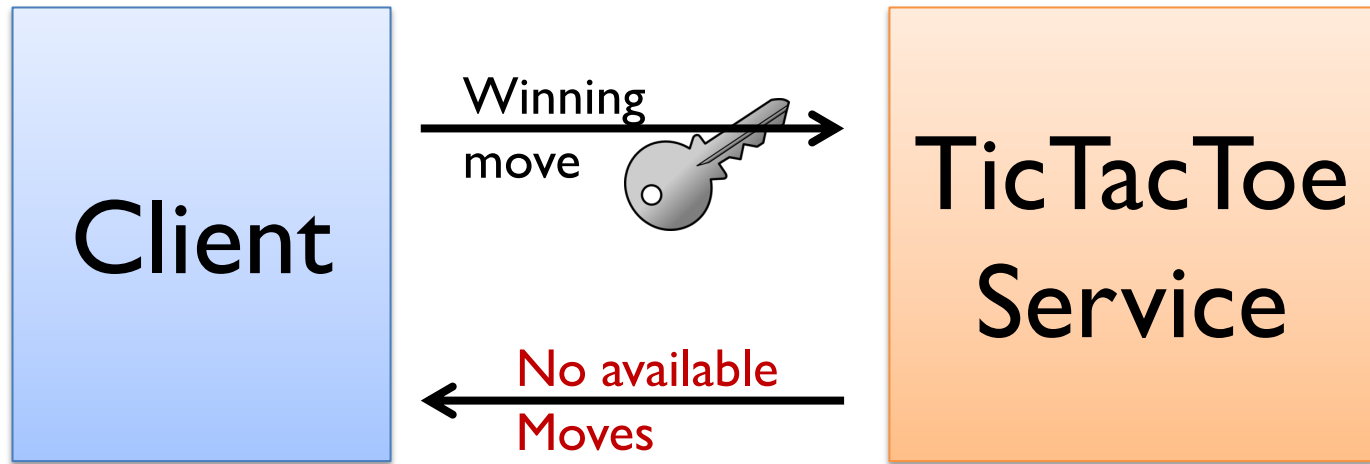
# Tic-Tac-Toe service with capabilities



# Tic-Tac-Toe service with capabilities



# Tic-Tac-Toe service with capabilities



# Tic-Tac-Toe API (cap-based version)

```
type MoveCapability =  
  unit -> TicTacToeResponse  
      // aka Func<TicTacToeResponse>
```

```
type MoveResult =  
  | KeepPlaying of MoveCapability list  
  | GameWon of Player  
  | GameTied
```

# Tic-Tac-Toe API (cap-based version)

```
type MoveCapability =  
  unit -> TicTacToeResponse  
      // aka Func<TicTacToeResponse>
```

```
type MoveResult =  
  | KeepPlaying of MoveCapability list  
  | GameWon of Player  
  | GameTied
```



Response contains all  
available moves

An intention-revealing interface

# Tic-Tac-Toe API (cap-based version)

```
type MoveCapability =  
  unit -> TicTacToeResponse  
      // aka Func<TicTacToeResponse>
```

The entire API!

```
type MoveResult =  
  | KeepPlaying of MoveCapability list  
  | GameWon of Player  
  | GameTied
```

```
type InitialMoves = MoveCapability list
```

Where did the "request" type go?  
Where's the authorization?

# Demo:

# Capability-based Tic-Tac-Toe



# What kind of errors can happen?

- ~~A player can play an already played move~~
- ~~A player can play twice in a row~~
- ~~A player can forget to check the response and keep playing~~

All fixed now! 😊

Is this good security or good design?

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 can guess the API  
you're doing it wrong

Security problem!

Also, a design problem —  
too much coupling.

# How to do HATEOAS

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



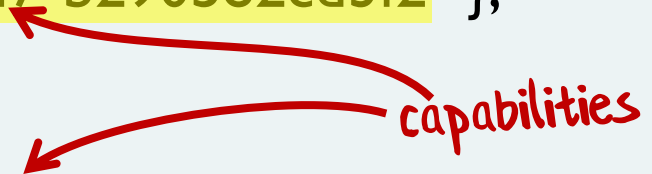
You can only know what URIs  
to use by parsing the page

Each of these URIs is a capability

# Tic-Tac-Toe HATEOAS

```
[  
  { "move": "Play (Left, Top)",  
    "rel": "Left Top",  
    "href": "/move/ec03def5-7ea8-4ac3-baf7-b290582cd3f2" },  
  { "move": "Play (Left, Middle)",  
    "rel": "Left Middle",  
    "href": "/move/d4532ca0-4e61-4fae-bbb1-fc11d4e173df" },  
  { "move": "Play (Left, Bottom)",  
    "rel": "Left Bottom",  
    "href": "/move/felbfa98-e77b-4331-b99b-22850d35d39e" }  
  ...  
]
```

*capabilities*



*An intention-revealing interface*

# Demo:Tic-Tac-Toe HATEOAS

Good security  $\Rightarrow$  Good design

Good design  $\Rightarrow$  Good security

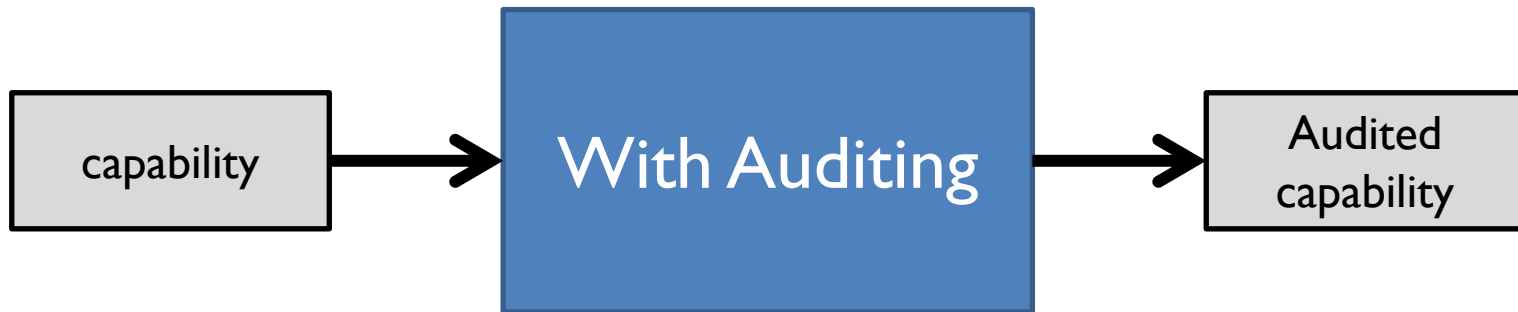
# **TRANSFORMING CAPABILITIES FOR BUSINESS RULES**

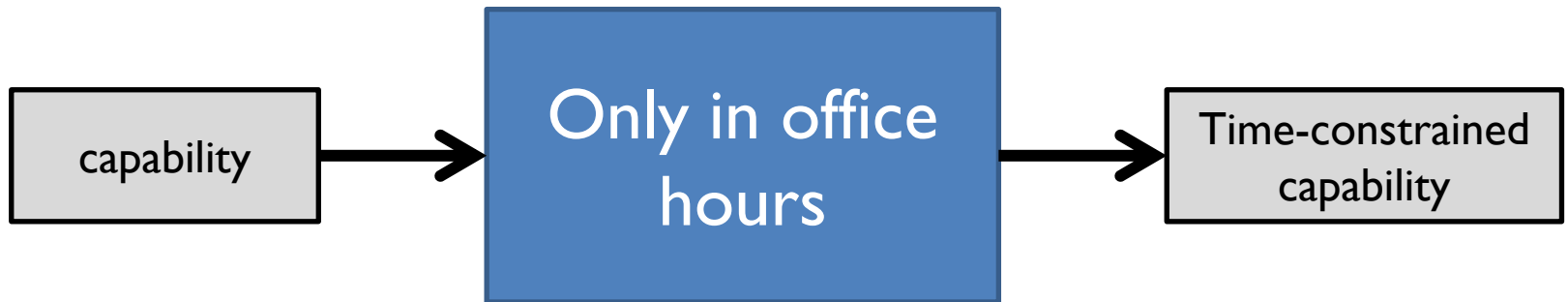


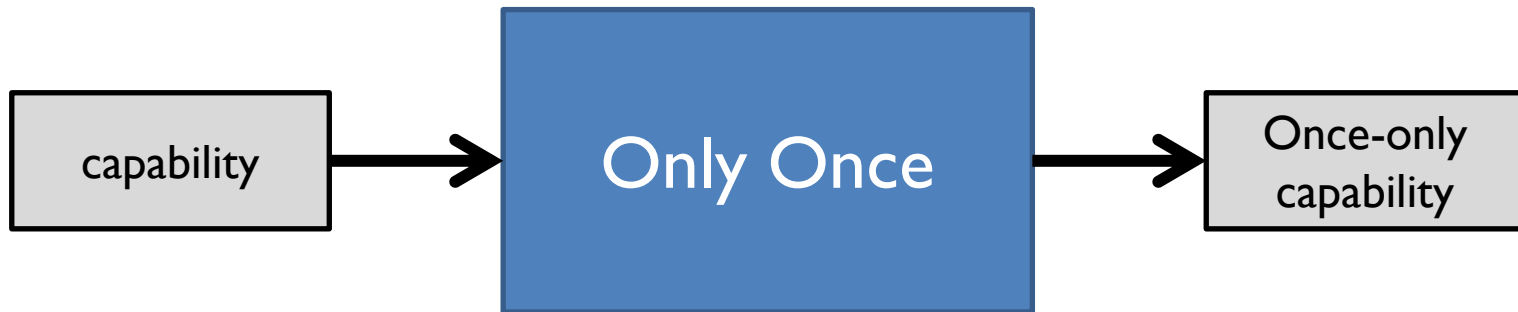
Capabilities are functions...

...so can be transformed to  
implement business rules









# How to revoke access in a cap-based system?

It's hard to revoke physical keys  
in the real world...

But this is software!

